INTRODUCTION TO COMPUTER ORGANIZATION AND DATA STRUCTURES: PDP–11 EDITION

COMPUTER SCIENCE SERIES

INTRODUCTION TO COMPUTER ORGANIZATION AND DATA STRUCTURES: PDP-11 EDITION

McGRAW-HILL COMPUTER SCIENCE SERIES

RICHARD W. HAMMING
Bell Telephone Laboratories

EDWARD A. FEIGENBAUM
Stanford University

Bell and Newell *Computer Structures: Readings and Examples*
Cole *Introduction to Computing*
Donovan *Systems Programming*
Gear *Computer Organization and Programming*
Givone *Introduction to Switching Circuit Theory*
Hamming *Computers and Society*
Hamming *Introduction to Applied Numerical Analysis*
Hellerman *Digital Computer System Principles*
Hellerman and Conroy *Computer System Performance*
Kain *Automata Theory: Machines and Languages*
Kohavi *Switching and Finite Automata Theory*
Liu *Introduction to Combinatorial Mathematics*
Madnick and Donovan *Operating Systems*
Manna *Mathematical Theory of Computation*
Newman and Sproull *Principles of Interactive Computer Graphics*
Nilsson *Artificial Intelligence*
Ralston *Introduction to Programming and Computer Science*
Rosen *Programming Systems and Languages*
Salton *Automatic Information Organization and Retrieval*
Stone *Introduction to Computer Organization and Data Structures*
Stone and Siewiorek *Introduction to Computer Organization and Data Structures: PDP-11 Edition*
Tonge and Feldman *Computing: An Introduction to Procedures and Procedure-Followers*
Tremblay and Manohar *Discrete Mathematical Structures with Applications to Computer Science*
Watson *Timesharing System Design Concepts*
Wegner *Programming Languages, Information Structures, and Machine Organization*
Winston *The Psychology of Computer Vision*

INTRODUCTION TO COMPUTER ORGANIZATION AND DATA STRUCTURES: PDP-11 EDITION

HAROLD S. STONE
Department of Electrical
and Computer Engineering
University of Massachusetts

DANIEL P. SIEWIOREK
Departments of Computer Science
and Electrical Engineering
Carnegie-Mellon University

McGRAW-HILL BOOK COMPANY

New York • St. Louis • San Francisco • Auckland
Düsseldorf • Johannesburg • Kuala Lumpur • London
Mexico • Montreal • New Delhi • Panama • Paris
São Paulo • Singapore • Sydney • Tokyo • Toronto

INTRODUCTION TO COMPUTER ORGANIZATION AND DATA STRUCTURES: PDP-11 Edition

Copyright © 1975 by McGraw-Hill, Inc. All rights reserved. Printed in the United States of America. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher.

34567890 DODO 798

This book was set in Modern 8A by Monotype Composition Company, Inc. The editors were Kenneth J. Bowman and Claudia A. Hepburn; the designer was Joseph Gillians; the production supervisor was Dennis J. Conroy. New drawings were done by ECL Art Associates, Inc.
R. R. Donnelley & Sons Company was printer and binder.

Library of Congress Cataloging in Publication Data

Stone, Harold S date
Introduction to computer organization and data structures, PDP-11 edition.

1. PDP-11 (Computer)—Programming. I. Siewiorek, Daniel P., joint author. II. Title.
QA76.8.P2S76 001.6'42 74-26622
ISBN 0-07-061720-1

FOR JAN, KARON, AND JEREMY

"And still they gazed, and still the wonder grew
That one small head could carry all he knew."
Goldsmith

"A little learning is a dangerous thing;
Drink deep, or taste not the Pierian spring:
There shallow draughts intoxicate the brain,
And drinking largely sobers us again."
Pope

CONTENTS

PREFACE xi
TO THE TEACHER xiii

1 ALGORITHMS, TURING MACHINES, AND PROGRAMS 2

1.1 Algorithms 4
1.2 Turing Machines 8

2 BASIC ASPECTS OF COMPUTER ORGANIZATION 16

2.1 Computer Memories 18
2.2 Structure of the Processing Unit 22
2.3 The Stored Program 25
2.4 Number Representations Based on a Binary Radix 30
2.5 Representation of Negative Numbers 31
2.5.1 Two's-Complement Representation 31
2.5.2 Ones'-Complement Representation 37
2.5.3 Signed-Magnitude Representation 41
2.5.4 Excess-2^{m-1} Representation 42
2.6 Floating-Point Representation 42
2.7 Representation of Instructions in Computer Memory 46

3 IBM SYSTEM/360-370 COMPUTER ORGANIZATION 56

3.1 Memory Organization 58
3.2 Organization of the Processing Unit 58
3.3 Arithmetic Instructions 60
3.4 Conditional and Unconditional Branching 65
3.5 Instruction Encoding 69
3.6 The Program Status Word 70

4 ADDRESSING MODES AND LOGICAL OPERATIONS 72

4.1 Addressing Modes 74
4.1.1 Relative Addressing 74
4.1.2 Register Addressing 76
4.1.3 Deferred Addressing 76
4.1.4 Additional Addressing Modes 79
4.1.5 Index Registers and Base Registers 86
4.1.6 Paging 90
4.2 Logic Operations 94
4.3 Shift Operations 97
4.4 Classification of Computers 104
4.4.1 Three-Address Computers 104
4.4.2 Two-Address Computers 105
4.4.3 One-Address Computers 106
4.4.4 Plus-One-Address Computers 106
4.4.5 Comparison of Addressing Schemes 108

5 ASSEMBLERS 112

5.1 Assembly Language 114
5.2 Assembly Language for System/360-370 124
5.3 The Two-Pass Assembler 128
5.4 Loaders and Program Relocation 133
5.5 Macro-Instructions 139

6 DATA STRUCTURES 146

6.1 One-dimensional Arrays 148
6.2 Multidimensional Arrays 155
6.2.1 The Address Polynomial Method of Indexing 155
6.2.2 Addressing by Indirection 156
6.3 Queues 158
6.4 Push-Down Stacks 162
6.5 Packed Data Structures 165

7 SUBROUTINES, COROUTINES, AND PARAMETERS 172

7.1 Subroutines 174
7.2 Coroutines 179
7.3 Parameters 186
7.3.1 The Mechanics of Passing Parameters 187
7.3.2 Call by Name and Call by Value 194

8 INPUT/OUTPUT OPERATIONS 204

8.1 Fundamentals 206
8.1.1 Paper-Tape Reader 209
8.1.2 Paper-Tape Punch 210
8.1.3 Teletype Input/Output 211
8.2 Interrupts 212
8.3 Control of Multiple Devices 220
8.4 Direct Memory Access Channels 227
8.5 Asynchronous Control of Input/Output 234

9 LINKED DATA STRUCTURES 252

9.1 Simple Lists 254
9.2 Two-Way Linked Lists 258
9.3 Threaded Lists 260
9.4 Memory Allocation and Garbage Collection 263

10 ROOTED TREES, RECURSIVE PROGRAMS, AND PUSH-DOWN STACK MACHINES 266

10.1 Rooted Trees 268
10.2 Traversing Binary Trees 271
10.3 Recursive Programs 279
10.4 Stack Instruction Sets 287

11 SORTING AND SEARCHING 298

11.1 Sorting 300
11.1.1 The Bubble Sort 300
11.1.2 Tree Sort 307
11.2 Search Algorithms 313
11.2.1 The Binary Search 313
11.2.2 Hash Addressing 316
11.2.3 Height-balanced Trees 326

APPENDIX 341
BIBLIOGRAPHY 351
INDEX TO IBM SYSTEM/360-370 INSTRUCTIONS 355
INDEX TO DEC PDP-11 INSTRUCTIONS 358
SUBJECT INDEX 361

PREFACE

The development of computers and computer programming over two decades has brought with it large programming systems, particularly compilers and executive programs, of enormous complexity. The construction of such systems would have been an impossibility without the concurrent development of the discipline of computer science. In the infancy of computers, computer science was hardly more than "a bag of tricks," a collection of clever but unrelated techniques devised by good programmers. The problems that faced the computer pioneers were usually "existence" problems; that is, is there an algorithm that computes X?

With time, the problems changed from existence problems to those of good design and practical application. Now we seek the best algorithms for computing X, where we measure the quality of an algorithm in terms of its storage requirements and execution time. The body of knowledge that constitutes computer science today is no longer an unorganized bag of tricks; rather, it is a highly structured discipline that includes tools for analysis and synthesis of programs.

The purpose of this text is to introduce the student to the most primitive actions of a computer and then show how the primitive actions can be put together to construct most of the complex actions that computers regularly perform. This text takes the student through an introductory treatment of Turing machines, into machine and assembly languages, number representation, and elementary programming. Data structures and input/output programs are the major concerns of the central portion of the text, and the concluding chapter develops techniques for analysis of programs through examples of algorithms for searching and sorting.

One of the primary objectives of the latter chapters is to show how the representation of data strongly influences both time and space requirements for processing the data. A variety of data structures are

presented, together with an analysis of their good and bad characteristics and typical applications for which they are well suited.

The scope of the text is sufficiently broad to cover most of Course B2, Computers and Programming, in ACM's Curriculum 68 as well as roughly half the material in Course I1, Data Structures. The material is intended primarily for college juniors and seniors with some computer experience. Very little mathematical background is needed, although a knowledge of elementary calculus and combinatorial analysis is helpful for understanding Chapter 11. Because college freshmen have also used this text successfully, it is likely that in time it can be used at the freshman level as more and more students receive their first exposure to computers before entering college.

This book reflects the contributions of many people. The authors express their appreciation to these people in general and in particular to Donald Rudisill, James Russ, Donald Thomas, and John Wakerly for their comments, suggestions, and encouragement. We are also indebted to Professor Angel Jordan and Professor John Grason for providing the PDP-11 facility that made the work possible.

Thanks are due Rita McVay, Geraldine Rutledge, Mary Ellen Sperow, and Susan Steidel for their meticulous care in typing the manuscript. Dorothy Josephson deserves a special note of acknowledgment for typing the many revisions required to make a final manuscript.

Harold S. Stone
Daniel P. Siewiorek

TO THE TEACHER

The material in the text is suitable for a semester course in machine language programming and data structures. Prerequisites ideally should include at least one course in computer programming with a compiler language such as FORTRAN, ALGOL, or PL/I. An ALGOL-like language is used throughout the text for descriptive purposes, but specific knowledge of ALGOL is not necessary. The descriptive language is developed in stages, so that the student learns the language along with the central course material. A short description of this language is given in the appendix. Calculus and statistics are desirable prerequisites, but not necessary. Only the material in the last chapter depends on these subjects. The mathematically oriented aspects of the last chapter can be skipped if necessary without marring the presentation of the search and sort algorithms.

The Digital Equipment Corporation PDP-11 and IBM System/360-370 computers are used as running examples throughout the text. It is clearly advantageous for students to have access to either of these computers for programming assignments. Initially the PDP-11 is described as a single accumulator machine with relative addressing. This subset of the PDP-11 architecture serves to familiarize the student with the fundamental operations of a computer. Subsequently more aspects of the PDP-11 architecture are introduced and the ALGOL-like description is updated to reflect the new additions. Subject matter is developed in a general framework and is made specific through examples for the PDP-11 and System/360-370. This approach not only allows the students to compare and contrast programming techniques on two different architectures but also allows them to apply their knowledge to computers not described in this text.

At Carnegie-Mellon University all programming assignments are done on a PDP-11, and each student operates the computer from the console. In this environment the students gain insight and learn to ap-

preciate many aspects of the computer that are hidden from them in a batch or time-sharing environment. This is essentially the only environment in which it is possible to construct interrupt-driven input/output programs. Hands-on computer facilities are somewhat of a luxury, however, and so it is likely that the usual programming environment will be time-sharing or batch processing. For either environment the early assignments should be based on the most primitive facilities. For example, in a time-sharing environment, a console linked to the computer can be made to appear to be the operator's console of a stand-alone computer. The students ought to be able to simulate the actions of pressing buttons and reading and writing the contents of memory cells in octal or hexadecimal representation. The objective here is to give them their first, and possibly their last, opportunity to program the computer directly, without the intervening levels imposed by compilers, link editors, loaders, and executive programs.

For any particular programming environment, this text must be supplemented with one or more reference manuals. For PDP-11 installations we recommend the *PDP-11 Processor Handbook*, published by Digital Equipment Corporation, Maynard, Massachusetts. Care should be taken to select the version that covers the particular model of the PDP-11 to be used by the students. The *PDP-11 Programming Card* summarizing the PDP-11 instruction set and reserved memory locations is also helpful to beginning students. Software can vary extensively between installations. This book generally assumes a minimal paper tape software system consisting of editor, absolute assembler, and absolute loader. Occasional reference is made in the text to features of the relocatable assembler and linker, or the macro-assembler. A list of software systems is available in the *PDP-11 Software Price List*. Reference manuals for the software available at a particular installation should also be made available to the students.

For System/360-370 computers the students should obtain *Principles of Operation*, Form A22-6821, and *Assembler Language*, Form C28-6514, both of which are available as part of the IBM System Reference Library. The instructor may have to supplement this reference material with a description of the input/output subroutines in use at the instructor's installation.

The programming assignments should be designed to introduce one new concept at a time during the course. A particularly effective first assignment, suggested by William McKeeman, is to have the students enter a short program at the console using the bit switches. The program is to present interesting patterns on the console lights as determined by the setting of the bit switches, perhaps in the context of a game. The "data" lights on most models of the PDP-11 display the data bus while

the computer is executing a program thus making dynamic patterns difficult to program. However, general register zero is displayed when the computer is halted allowing static patterns by loading register zero and effectively halting the computer (for example, by using the HALT and RESET instructions). If the data and address share the same set of console lights, the above behavior does not hold. To check the console operation of a particular model of the PDP-11 see the *PDP-11 Processor Handbook*. This initial assignment can be done after only a week of instruction in basic principles. An assignment in assembly language requires one to two weeks more preparation and is considerably more difficult as a first assignment than as a second assignment.

A typical set of program assignments for a one-semester course are:

1. Write a short program that displays interesting patterns on the console lights. Enter this program by means of the console bit switches.
2. Write a nonrelocatable program in assembly language. Load and execute the program. The program should take input from the console switches and produce an output in the registers or through an automatic dump. Multiplication and division algorithms, simple numeric processing, and number conversion are suitable types of programs.
3. Write two or more relocatable programs that have to be linked to each other. Load the programs with a linking loader and execute them. This assignment can make use of library subroutines for input/output. It should cover one or more of the data structures in Chapter 6.
4. Write an input/output program that uses wait loops. This program can do simple operations on input data before transmitting it to the output device. This assignment can also cover one or more of the data structures in Chapter 6.
5. Write an interrupt driven input/output program. This assignment covers material in Chapters 7 and 8.
6. Write a program that demonstrates some nontrivial process on linked data structures.
7. Write a program that performs Floyd's tree sort algorithm.
8. Write a program for searching with hash addressing.

To use this book for a one-quarter course, instead of a one-semester

course, Chapter 1 may be omitted and the material in Chapters 9 to 11 may be treated selectively. The core material for one quarter is in Chapters 2 to 8.

A hands-on minicomputer to support this text should be configured as indicated below:

1. Processor with a minimum of 4K words, preferably 8K words of storage
2. Teletype or console typewriter
3. Card reader or modem for transporting programs between the stand alone minicomputer and a central computer system
4. Auxiliary memory for system program storage (cassette tape or disk)
5. Input/output devices for storing and reloading student programs (paper-tape reader and paper-tape punch or cassette tape)
6. A line printer or similar device for program listing

The line printer is not a necessity in many university environments, since programs can be assembled and listed on a central facility even when they are to be executed on a stand-alone minicomputer. There are several versions of an assembler for the PDP-11 that executes on System/360-370. Initial program creation and assembly on the System/360-370 can relieve some editor/assembler congestion on the minicomputer. Similar assemblers that also execute on System/360-370 are available for other computers.

Harold S. Stone
Daniel P. Siewiorek

INTRODUCTION TO COMPUTER ORGANIZATION AND DATA STRUCTURES: PDP-11 EDITION

ONE

ALGORITHMS, TURING MACHINES, AND PROGRAMS

One of the major purposes of this book is to investigate the typical primitive instructions that are found in computers and to explore techniques for using sequences of such instructions to do computations. In this chapter we introduce the mathematical foundations of computation through abstract models of computing devices. In later chapters we shall turn our attention to real computers.

1.1 ALGORITHMS

Let us adopt the point of view that a computer is a robot that will perform any task that can be described as a sequence of instructions. In this section we shall focus attention on the properties of sequences of instructions that can be obeyed by a computing robot.

Within the acceptable set of instructions are those which tell the robot to do simple arithmetic operations such as addition, subtraction, multiplication, and division. Moreover, the instructions must permit conditional actions to be taken by the robot, as in the instruction "Divide by A if A is not equal to 0."

In order to give the widest possible flexibility in forming acceptable instruction sequences, we allow the robot to obey instructions that cause it to obey other instructions. For example, instead of obeying the sequence of instructions

1. Output the number 0.
2. Output the number 0.
3. Output the number 0.
4. Halt.

the robot could obey the equivalent instructions

1. Repeat instruction A three times.
2. Halt.

A. Output the number 0.

A sequence of instructions of this type, or in fact any sequence of instructions that can be obeyed by a robot, is called an *algorithm*. Our discussion thus far suggests the following slightly more formal, but tentative, definition.

Definition An *algorithm* is a set of rules that precisely define a sequence of operations.

Although this definition of algorithm is rather vague, it is useful for introducing the concept of algorithms. Later we shall proceed to a more rigorous mathematical definition.

It is often the case that algorithms are directed to people rather than robots. For people to follow the rules of an algorithm, the rules must be formulated so that they can be followed in a robot-like manner, that is, without the need for thought. For example, the following set of rules is an algorithm for finding the roots of the equation $ax^2 + bx + c = 0$:

1. The roots of the equation

   $$ax^2 + bx + c = 0$$

   are found by evaluating the expressions

   $$\frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

   and

   $$\frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

   where $a \neq 0$.

2. If $a = 0$, then the equation has at most one root, which may be found by evaluating the expression

   $$\frac{-c}{b}$$

   where $b \neq 0$.

3. If both $a = 0$ and $b = 0$, then the equation has no roots.

Notice that all the operations that we have to perform to find the roots of a quadratic equation seem to be well defined, since they involve only the usual simple arithmetic operations, the ability to determine if a number is 0, and the operation of taking a square root. However, if the instructions are to be obeyed by someone who knows how to perform arithmetic operations but does not know how to extract a square root, then we must also provide a set of rules for extracting a square root in order to satisfy the definition of algorithm.

Cooking recipes, travel directions, and college registration instructions are also algorithms. Sometimes, however, these examples do not satisfy our definition of algorithm, for they may include questionable instructions such as "use a handful of flour" or "turn a mile after the last bridge"; you should have no difficulty in finding examples of college registration instructions that fail to satisfy our definition.

Algorithms, as pointed out earlier, may specify iterative behavior in that a particular instruction may be carried out several times. Moreover, the number of times that an operation is to be performed need not be stated explicitly. It may depend on the particular context in which the algorithm is applied. Consider the word description of a sorting algorithm in Example 1.1.

EXAMPLE 1.1

To sort a list of numbers into ascending numerical order, proceed by scanning the list from beginning to end. Each step of the scan consists of inspecting a pair of adjacent numbers in the list. If the first number of a pair is greater than the second number, then the two are interchanged. After inspecting a pair of numbers, and interchanging if necessary, the scan of the list continues by examining the pair of adjacent numbers beginning with the second number of the pair that has just been examined.

Scans of the list are repeated until a scan is performed during which no interchanges occur.

The sorting algorithm of Example 1.1 does not specify how many times to scan the list. If the list contains n numbers, then what is the most number of times that the list will be scanned? What is the fewest?

Clearly, not all instructions are acceptable, because they may require the robot to have abilities beyond those that we consider reasonable. For example, we may instruct a computing robot to "Output the number 0 if Henry VIII was a King of England, otherwise output the number 1." Intuitively we should not expect a robot to obey that instruction unless it has somehow previously been instructed to store the fact that Henry VIII was a King of England. It is even less reasonable to expect the robot to respond to the instruction to "Output the number 0 if Aristotle was a King of England, otherwise output the number 1." In this case either the robot must specifically have the information that Aristotle was not a King of England, or it must have the names of all English kings so that it can search the list of names for the name Aristotle. Suppose the robot has the names of only five English kings. If it cannot find Aristotle's name on that list, what should it do? Obviously in this case the robot cannot determine whether Aristotle was an English king, and so it cannot obey the instruction.

The problem exemplified above is that the instruction requires the

robot to have particular information, so that if the information is not given to the robot, it cannot obey the instruction. It may guess what to do, of course, but guessing cannot be permitted if we are to be sure that the robot performs computations correctly. An intuitive definition of an acceptable sequence of instructions is one in which each instruction is precisely defined so that the robot is guaranteed to be able to obey it. With this definition the King of England instruction could be included in an instruction sequence only if the robot were given sufficient information about English kings to determine how to obey the instruction. We say that an instruction is *effective* if there is a procedure that the robot can follow in order to determine precisely how to obey the instruction. In our example the Aristotle instruction is effective when the robot has a list of all Kings of England, but if the list of Kings of England is incomplete, the instruction is not effective.

The effectiveness of an instruction is sometimes dependent on a finiteness property. For example, consider the instruction "If every map can be colored with no more than four colors so that no two adjacent countries have the same color, then output the number 1, otherwise output the number 0." This instruction embodies the famous *four-color conjecture*, which has challenged mathematicians for over a century. Either every map can be colored with four or fewer colors and satisfy the adjacent-country constraint, or there is some map that requires at least five colors to satisfy the constraint. No one has been able to find a map that needs five colors, nor has anyone proved that four colors are sufficient for all maps. Our computing robot could consider every possible map with one country, then two countries, then three countries, etc., and consider every possible way of coloring each of these maps with four or fewer colors. Since there is only a finite number of different ways that n countries can share boundaries, and since for each map with n countries there is only a finite number of different ways it can be colored with four or fewer colors, if there is a map that requires five colors, our robot will surely find it. So our robot can definitely output the number 0 if the four-color conjecture is false. Conversely, if the conjecture is true, then our robot will never discover this. Since the procedure we have given is not guaranteed to terminate in a finite number of steps, the four-color instruction is not an effective instruction.

With a slight modification we can form a similar instruction that is effective because it is finite. Consider the instruction "If every map with fewer than 26 countries can be colored with no more than four colors so that no two adjacent countries have the same color, output the number 1, otherwise output the number 0." Since we have limited the number of countries to 26 or fewer, our computing robot can examine every possible way that 26 or fewer countries can share boundaries and can decide if the

four-color conjecture is true for 26 or fewer countries. Clearly the new instruction is effective, and it is effective if we replace the number 26 by any integer we care to name.

The preceding discussion illustrates that every algorithm must have the property of *finiteness;* that is, it must terminate in a finite number of steps. The first of the four-color procedures above is guaranteed to settle the four-color conjecture, because each map will eventually be examined unless the robot finds a counterexample to the conjecture. It could be shown by a tedious exercise that each instruction in the procedure is both finite and effective; yet the procedure as a whole is not an algorithm because it will not terminate in a finite time if the conjecture is true.

There is one more property that we require of algorithms. Each instruction of an algorithm must be precisely defined so as to avoid any ambiguity or misinterpretation. Algorithms that satisfy this property are said to be *definite.* An example of a definite instruction is "If x is greater than 1, then output 1, otherwise output 0." Suppose, however, that the instruction is "If the number x is very much greater than 1, then output 1, otherwise output 0." Now, we know that the robot must output 0 if $x = -2$, and we feel rather sure that it will output 1 if $x =$ 1,000,000. However, what should it do if $x = 17$? And if it produces a 0 when $x = 17$, should it also produce a 0 when $x = 343$?

To summarize the discussion in this section, we define an algorithm to be a set of rules that precisely defines a sequence of operations such that each rule is effective and definite and such that the sequence terminates in a finite time.

1.2 TURING MACHINES

In order to make the definition of algorithm more rigorous let us consider the *Turing machine.* This "machine" is an abstract mathematical model that takes its name from A. M. Turing, who first introduced the concept in 1936 in a paper dealing with the theory of computation.

A Turing machine algorithm will be composed of rules drawn from a very restricted set. We shall suppose that the rules are obeyed by a computing device called a Turing machine. Since the actions of the Turing machine are motivated by the actions of a human being in carrying out an algorithm, the Turing machine is given "scratch paper" in the form of an infinite tape such that each square on the tape can contain only one symbol from a finite alphabet of symbols.

The Turing machine can read and write on its tape by means of a *head.* At any instant of time the head can read or write a symbol only in the square of the tape that is directly under it, but the Turing machine

can be instructed to move the head left or right along the tape, one square at a time, to place any square under the head. Besides the tape memory, the Turing machine contains one other cell. This cell holds a symbol called the *state* of the machine, and this symbol is drawn from a finite alphabet that is different from the tape alphabet. An algorithm for the Turing machine consists of a set of rules that involve head motion, tape operations, and change of state. The state of the machine at a particular instant of time identifies the instruction that the Turing machine is carrying out at that time. Figure 1.1 shows a conceptual diagram of a Turing machine, with the head represented by an arrow pointing to a square on the tape and the state of the machine shown as a number attached to the head.

The actions of the Turing machine are determined completely by rules that specify the actions to be taken for each possible state and each possible observed symbol.

To be more precise, each rule for a Turing machine is a five-tuple of the form

$$(q_i, s_j, q_{ij}, s_{ij}, d_{ij})$$

where q_i is one of the states of the machine and s_j is one of the symbols that may appear on the tape of the machine. For each state-symbol pair the rule gives the next state of the machine, q_{ij}; the symbol to write in the currently observed cell, s_{ij}; and the next cell to observe on the tape, d_{ij}. By convention, $d_{ij} = 1$ if the next cell is the cell to the right of the current cell, -1 if it is the cell to the left, and 0 if the current cell will be observed again as the next cell. Also by convention, one of the states is distinguished as the stopping state and is given the name HALT.

An algorithm for the Turing machine consists of a set of five-tuples such that there is exactly one five-tuple for each state-symbol pair, excluding the HALT state. Such an algorithm is known as a *Turing machine program*.

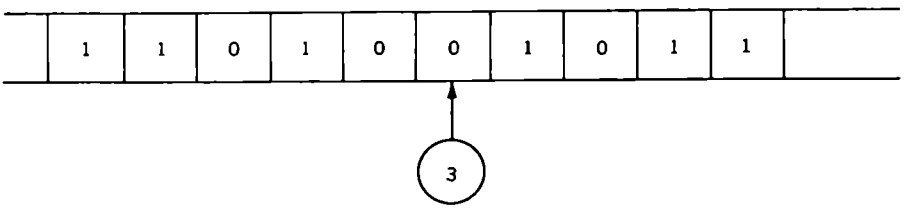


FIGURE 1.1 A conceptual view of a Turing machine

The computation of a Turing machine is described by stating:

1. The tape alphabet
2. The form in which parameters are represented on the tape
3. The initial position of the head of the Turing machine
4. The initial state of the Turing machine
5. The form in which answers will be represented on the tape when the Turing machine halts
6. The machine program

EXAMPLE 1.2

The following machine adds 1 to a positive integer and then halts.

Tape alphabet: 0,1.

Input representation: The integer i is represented by a string of i adjacent 1's bounded by 0's.

Head position: On the leftmost 1 of the input parameter if the parameter is nonzero, otherwise on a 0.

Initial state: State 1.

Output representation: The integer is represented by a string of adjacent 1's bounded by 0's.

Machine program:

q_i	s_j	q_{ij}	s_{ij}	d_{ij}
1	0	HALT	1	0
1	1	1	1	1

Example 1.2 illustrates a very simple Turing machine algorithm that has one state other than the HALT state. This algorithm adds 1 to an integer by the following strategy. The representation of an integer is a *unary* representation; that is, the integer i is represented by i consecutive 1's bounded on each side by 0's. To add 1 to an integer the head is moved right until a 0 is found. The machine replaces the 0 by a 1 and then halts. The algorithm works equally well if the head is moved left instead of right.

EXAMPLE 1.3

The following machine compares two positive integers for equality.

Tape alphabet: 0,1.

Input representation: Unary (same as Example 1.2), two numbers placed on tape separated by a single zero.

Head position: On the 0 between the two numbers.

Output representation: The head is reading a 0 if equal and a 1 if unequal.

Initial state: State 1.

Machine program:

q_i	s_j	q_{ij}	s_{ij}	d_{ij}	ACTION
1	0	2	0	1	Move to right of center mark.
1	1	—	—	—	Impossible state-symbol pair.
2	0	3	0	−1	No 1's to right of center, look to left of center.
2	1	4	1	1	Found a 1 on right, move to end of string of 1's.
3	0	HALT	0	−1	Move left and point to answer.
3	1	—	—	—	
4	0	5	0	−1	At end, back up one square.
4	1	4	1	1	Scan over 1's.
5	0	—	—	—	
5	1	6	0	−1	Erase the rightmost 1.
6	0	7	0	−1	At center, move to check left parameter.
6	1	6	1	−1	Scan left to center marker.
7	0	HALT	1	0	Mark unequal and stop.
7	1	8	1	−1	Scan to leftmost 1.
8	0	9	0	1	Back up one square.
8	1	8	1	−1	Scan to leftmost 1.
9	0	—	—	—	
9	1	10	0	1	Erase leftmost 1.
10	0	1	0	0	Go back to start.
10	1	10	1	1	Scan right to center marker.

Example 1.3 is a Turing machine program for comparing two numbers for equality. The two numbers are placed on the tape initially so that they are separated by a single 0, and the algorithm begins with the head reading that 0. The strategy of this algorithm is to subtract 1 from each of the numbers until one of them becomes 0. The two numbers are equal if they both become 0 after the same number of 1's are subtracted from each. Notice in particular that the 1's are subtracted from the far right and far left on the tape. This guarantees that the two numbers are always separated by a single 0 until one of them becomes 0. If 1's were subtracted from the middle, then there would be a string of several 0's separating the numbers, and this string would grow in length as more 1's were subtracted. Consider what would happen in this program when the last 1 of a number were subtracted. The next time that the machine tried to subtract 1 from that number, it would start the head moving over the string of 0's looking for a 1, and the 1 would never appear, causing the machine to repeat the same instruction forever. Of course, there are ways of preventing this behavior if 1's are subtracted from the middle, but the point here is that there are a number of subtle aspects of Turing machine programs which are related to the problem of positioning the head on the tape. The strategy of subtracting the outermost 1's was selected for Example 1.3 primarily because it neatly disposes of the head-positioning problem.

It is rather interesting to investigate other computations that can be done by a Turing machine. Turing machines can be programmed to do addition, subtraction, multiplication, and division of numbers by using strategies similar to those of Examples 1.2 and 1.3. It is not surprising that they can perform more complex operations such as the extraction of the square root of a number, since such operations are based totally on the arithmetic operations and the ability to make conditional changes in an instruction sequence.

Turing machine algorithms that are guaranteed to halt in a finite time satisfy the definition of algorithm presented earlier. Since our definition did not specify the form of the rules of an algorithm, there is sufficient latitude in our definition to allow other abstract models of algorithms. As we examine the form of rules we may use for algorithms, we find it is very difficult to identify a set of rules that is truly different from the rules that define the behavior of Turing machines. For example, we may seek a set of rules that permit arithmetic operations such as addition and subtraction. Although Turing machines cannot perform arithmetic operations as single steps, they certainly can perform all of the arithmetic operations. Hence arithmetic operations alone cannot increase the power of computation of a set of rules beyond the power of the rules that govern the behavior of Turing machines.

Several mathematicians have proposed models of algorithms that use rules substantially different from the Turing machine rules. Nevertheless, all the abstract mathematical models have proved to be equivalent in that the sets of computations that can be described by algorithms within each model are exactly the same. The evidence seems to indicate that every algorithm for any computing device has an equivalent Turing machine algorithm. In fact, Alonzo Church has formulated a famous conjecture which says in essence that everything that is computable is computable by a Turing machine. So far Church's thesis has not been shown to be either true or false. If the conjecture is true, then it is certainly remarkable that Turing machines, with their extremely primitive operations, are capable of performing any computation that any other device can perform, regardless of how complex a device we choose.

REFERENCES

Readers interested in pursuing the study of the theory of computation will especially enjoy Minsky (1967), from which some of the material in this chapter is drawn. Minsky's book contains very good discussions of Church's thesis and noncomputable functions, among other topics. Davis (1958) is a much deeper text on this subject and may be of interest to the advanced student. Rogers (1967) contains a thorough discussion of Church's thesis and its importance in the study of recursive functions. The original definition of a Turing machine appeared in an article by Turing (1936).

EXERCISES

1.1 Several state-symbol pairs are not used at all in Example 1.3. Find a way to reduce the number of states by modifying the five-tuples. Simulate your program on a tape that contains the number 5 to the left of the marker and the number 3 to the right. Show the contents of the tape, the state of the machine, and the position of the head at just those instants when the head is about to change direction.

1.2 Write a Turing machine program that doubles an integer. Use a binary tape alphabet. Simulate your machine as described in Exercise 1, given the number 4 on its tape.

1.3 Write a Turing machine program that computes the product of a pair of operands. Use a four-symbol tape alphabet $\{0,1,X,Y\}$ and a unary representation of input and output data. Simulate your machine given the numbers 2 and 3.

1.4 Write a Turing machine program that compacts disjoint sequences of 1's into one contiguous sequence. Assume that the sequences to be compacted lie to the right of the initial tape head position, that they are separated by a single 0, and that at least two 0's lie to the right of the rightmost sequence. Simulate your program on the tape $\cdots 001101011100 \cdots$ and show that it produces the tape $\cdots 001111110000 \cdots$

1.5 Prove that the following Turing machine program will never reach the HALT state when started on the all-0's tape (Lin and Rado, 1965).

q_i	s_j	q_{ij}	s_{ij}	d_{ij}
1	0	2	1	1
1	1	HALT	1	1
2	0	3	0	−1
2	1	2	1	−1
3	0	1	0	1
3	1	2	1	−1

TWO

BASIC ASPECTS OF COMPUTER ORGANIZATION

The previous chapter describes the Turing machine as an abstract model of a computer. In this chapter we shall see that the Turing machine, in spite of its abstract qualities, has many characteristics in common with modern digital computers. Figure 2.1 shows a block diagram of a typical computer. The memory component is the counterpart of the Turing machine tape. The module labeled *processing unit* corresponds roughly to the portion of a Turing machine that executes instructions. The representation in Figure 2.1 is greatly oversimplified, primarily to emphasize the similarity of computers to the Turing machine. When we investigate the detailed behavior of computers, we shall see that computers are in fact somewhat different from the Turing machine.

The material that follows describes the characteristics of a typical digital computer, particularly its instruction repertoire, the instruction-execution mechanism, and the representation of numbers. The Digital Equipment Corporation PDP-11 serves as a running example throughout this chapter, but the treatment is sufficiently general to encompass a wide variety of computers.

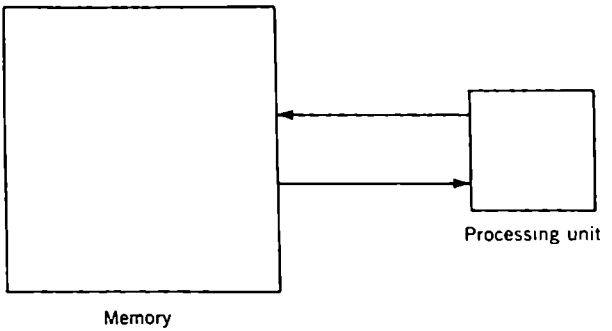


FIGURE 2.1 The structure of a digital computer

2.1 COMPUTER MEMORIES

One of the difficulties encountered in programming Turing machines is the problem of storing intermediate results on the tape and retrieving the information later. When information must be copied from a storage area on the tape to an active computation area of the tape, the Turing machine must follow a complex sequence of instructions to move the head back and forth between the two portions of the tape. The head might make several trips between the two tape areas, one trip for each symbol of information. Fortunately, advanced technology has provided storage media for computers that are far less cumbersome than Turing machine tapes. Digital computer memories provide access to any specified storage position in a single operation. Moreover, the time required for a memory operation is independent of the location of the last item accessed. This mode of access is known as *random access*.

Figure 2.2*a* shows, in block diagram form, the organization of a typical computer memory. Some computer memories are not organized precisely as indicated in the illustration or do not behave exactly according to the following description. Nevertheless, the presentation here is sufficiently accurate for our purposes.

The memory contains a storage area, two registers designated MDR (memory data register) and MAR (memory address register), and three control inputs designated GO/IDLE, READ/WRITE, and BYTE/WORD. The storage area is divided into cells called *bytes*, which are very much like the cells on the Turing machine tape. Each byte is uniquely identified by a number called the *byte address*. If the memory has N bytes, then the addresses are 0, 1, 2, . . ., N-1. Each byte has the ability to store one datum, which corresponds to the symbol stored in a cell on the Turing machine tape. The MDR and MAR registers are two special memory cells whose function is described below. We use the word *register* to identify a memory cell used for specialized functions, in contrast to a memory

cell used for general-purpose information storage in the main computer memory. In most computers, registers are constructed for higher-speed operation than the main computer memory, usually at a higher cost per component when the extra cost is justifiable. As we shall see, the speed of the entire computer depends critically on the speed at which we can modify the contents of the MDR and MAR registers.

The memory shown in Figure 2.2*a* looks somewhat like a Turing machine tape because it is one-dimensional, but there are essential differences. Naturally, real computers cannot have infinite memories, and so there is an upper limit on the amount of information that can be stored in a computer memory. No such limit exists for a Turing machine. The use of addresses for computer memories is another distinguishing feature. We can speak about the contents of "the byte with address 106"

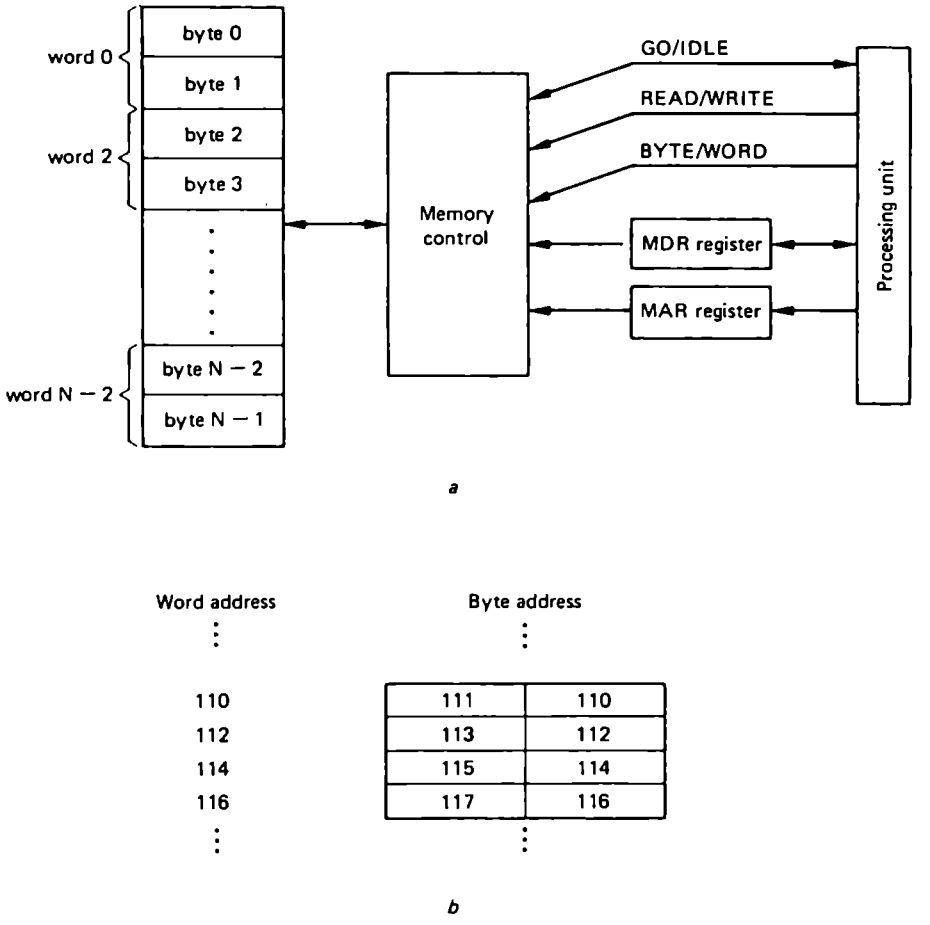


FIGURE 2.2 (*a*) The organization of a typical computer memory; (*b*) the organization of the PDP-11 storage area into words

in a computer memory, and the address distinguishes this byte from all the others. Since the Turing machine tape does not have an address associated with each cell, we can only identify a specific piece of information on the tape in terms of its surrounding environment. For example, we might refer to the "datum immediately to the right of the first pair of adjacent 0's that lies to the right of the present head position."

The GO/IDLE control line shown in Figure 2.2*a* controls the activity of the memory. It can be in one of two states that are designated GO and IDLE. When the control line changes state from IDLE to GO, the memory begins a cycle of operation. At the completion of an operation, the memory sets the control line to IDLE and awaits a new request.

Although the memory is divided into cells each of which contains a byte, the memory can operate in a mode in which a pair of adjacent bytes is accessed simultaneously during a single memory cycle. The pair of bytes must be such that the first byte has an even address and the second byte has the next greater address. A pair of bytes with this property is called a *word*. The address of the even byte is called the *word address*. Figure 2.2*b* depicts the storage area reorganized into words. The control line labeled BYTE/WORD controls the mode of memory access. In the BYTE mode, the memory accesses bytes; and in the WORD mode, the memory accesses words. In the following discussion, unless otherwise specified, we assume that all memory accesses involve words rather than bytes.

To store one word of information at a particular place in memory, the word address is placed in the MAR and the new information is placed in the MDR. The READ/WRITE control line is placed in the WRITE state; the BYTE/WORD control is set to WORD; and the GO/IDLE control is set to GO. (All these operations are performed by the processing unit of the computer.) When the memory senses the GO signal, it begins its operation cycle. The WORD signal on the BYTE/WORD control line indicates that the MDR contains a word of information and that the MAR contains the address of a word. The WRITE signal on the READ/WRITE control line causes the memory to erase the information found at the word address indicated by the MAR. The byte whose address corresponds to the word address is replaced by the information in the right half of the MDR, and the byte with the next higher address is replaced by the information in the left half of the MDR. The storing of a byte is similar to the storing of a word except that the BYTE/WORD control is set to BYTE. The addressed byte is replaced by the information in the right half of the MDR.

Retrieval of a word is performed by placing the word address in the MAR, setting the BYTE/WORD control line to WORD, and setting the READ/WRITE control line to READ. When the GO signal is issued, the

memory copies the word of information at the location given by MAR into the MDR register. The byte whose address corresponds to the word address is copied into the right half of the MDR, and the byte with the next higher address is copied into the left half of the MDR. The contents of the memory are not changed by this operation. The retrieval of a byte is similar to the retrieval of a word except that the BYTE/WORD control is set to BYTE. The addressed byte is placed into the right half of the MDR.

A more compact way of expressing the behavior of the memory is given by the following symbolism:

```
WRITE:   MEMORY[MAR+1;MAR] ← MDR;

READ:    MDR ← MEMORY[MAR+1;MAR];
```

The notational conventions used here follow those of the programming language ALGOL. The bracketed items are analogous to subscripts, so MEMORY[MAR+1;MAR] refers to the two adjacent bytes in memory starting at the address that is stored in the MAR register. Similarly, MEMORY[MAR] refers to a byte whose address is stored in the MAR register. The left arrow signifies data movement. When an item appears to the left of a ←, as does MEMORY[MAR+1;MAR] in the WRITE action, we treat it as the name of the destination of data. Hence MEMORY[MAR+1;MAR] in the WRITE action signifies that MEMORY[MAR+1;MAR] receives the data. When an item appears to the right of ←, it is treated as a value. Hence the MEMORY[MAR+1;MAR] that appears in the READ action refers to the contents of MEMORY[MAR+1;MAR].

Thus far we have not discussed the physical mechanisms involved in storage of data or the technical details of the design of a computer memory. A complete discussion is beyond the scope of this text, but some technological characteristics deserve comment because they influence the organization and the programming of computers.

Many computers use a magnetic medium for storing data in memory. The most common memory device in use today is the *magnetic core*, a small doughnut-shaped piece of ferrite with a diameter about one- or two-hundredths of an inch. When a core is magnetized, the magnetic domains can be aligned either clockwise or counterclockwise around the central hole. Thus a core can assume one of two magnetic states. By convention, one state is said to represent the symbol 0 and the other the symbol 1. A typical computer memory contains from 10^5 to 10^7 magnetic cores.

As technology has advanced, more and more computers now utilize semiconductor storage devices in their memories. Information is stored

either as electric charges or as current flows; each storage device in a semiconductor memory can be in one of two states depending on the presence or absence of an electric charge or current flow. In semiconductor memories, as in magnetic memories, one state represents a 0 and the other a 1. The advantage of a semiconductor memory over a magnetic memory is that data can be accessed in much less time. However, a semiconductor memory loses its stored information when power is turned off, whereas magnetic memory retains the information.

As mentioned previously, computer memory is organized into bytes, each of which can hold a single datum. Each byte contains a fixed number of cores, so that a datum stored at a byte in memory consists of a vector of 0's and 1's. The length of the vector is the number of cores per byte. The individual components of a datum are called *binary digits* or *bits*. The PDP-11 has 8 bits in each byte. Note that since the PDP-11 can operate in word mode, it is capable of accessing one word of 16 bits in a single operation. Often it is convenient to view a word as a single datum of 16 bits, rather than as a pair of 8-bit bytes of data.

In some computers, all data accesses deal with items containing a fixed number of bits. In such computers, a single datum is called a word, and the computers are said to be *word-oriented*. The number of bits per word varies from computer to computer and may be as small as 8 or as high as 64; multiples of 6 and 8 are most common. Some computers are similar to the PDP-11 in that data of two or more different lengths can be manipulated by the memory. If the smallest addressable item is a single bit, the computer is said to be *bit-oriented;* if it is an 8-bit datum, the datum is called a byte and the computer is said to be *byte-oriented*. Thus the PDP-11 is a byte-oriented computer. The smallest addressable item is rarely anything other than a bit, an 8-bit byte, or a word of the length for the computer in question.

The PDP-11 has two complete sets of instructions: one that manipulates words and one that manipulates bytes. Initially, our discussion will deal exclusively with word instructions. Byte instructions will be described in Chapter 6.

In a later section we shall discuss how numbers are represented and manipulated in a binary number system. For the present we shall continue to assume that each byte or each word in memory can hold a single number.

2.2 STRUCTURE OF THE PROCESSING UNIT

The *processing unit* is the portion of a computer that executes program instructions, and thus it corresponds to the part of a Turing machine that performs the actions specified by five-tuples. The processing unit contains a few special registers that are used for holding data in active

computation. Each processor register serves a particular function in the same manner that the MDR and MAR registers serve special functions. One such register, called the *P register* (for *program counter*), identifies the instruction currently being executed. The P register is thus the computer counterpart of the current state of a Turing machine.

The processing unit also contains one or more data registers that serve as temporary storage and are used for arithmetic processing of data. In particular, the DEC PDP-11 computer has an accumulator that can hold one word (two bytes). The accumulator is represented by the abbreviation AC. Figure 2.3 shows the gross structure of the processing unit and its interconnections with memory.

To illustrate the behavior of the processor we shall consider a few instructions in the instruction repertoire of the PDP-11 computer. Instructions for other computers are similar, but not necessarily identical, to those given here.

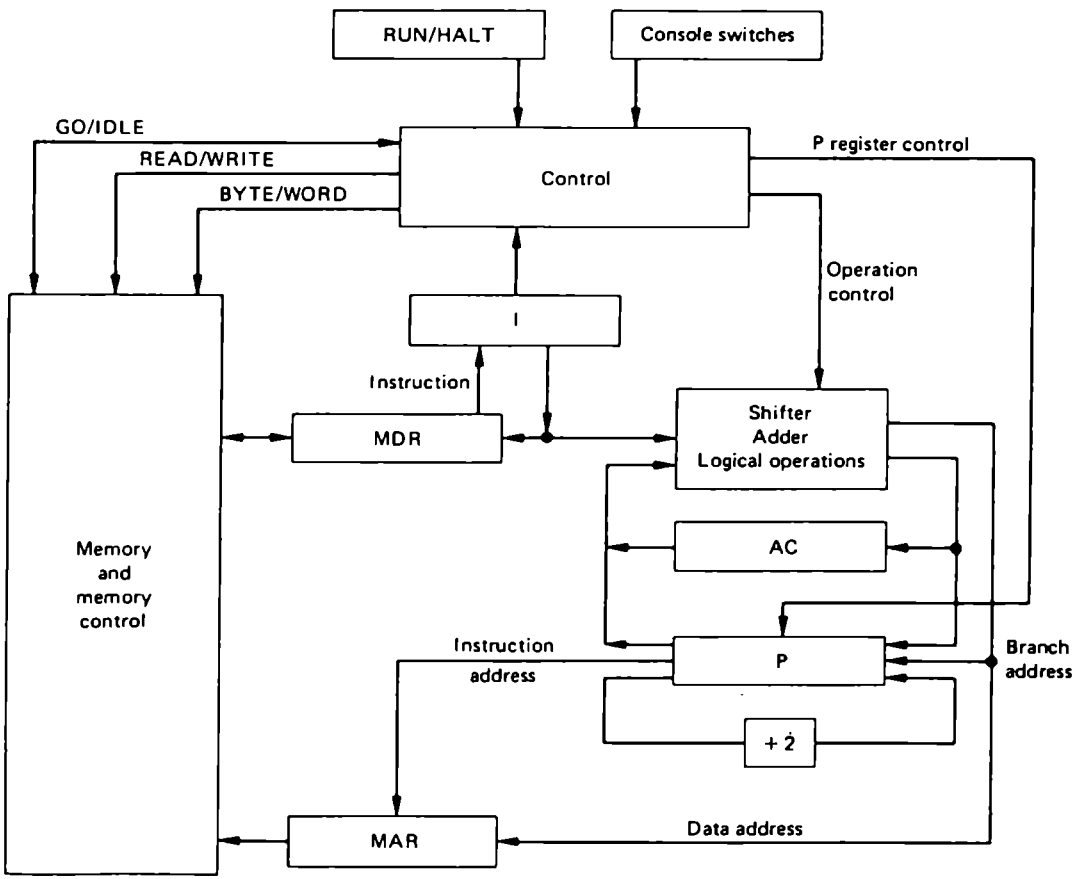


FIGURE 2.3 The structure of a typical processing unit

Each instruction described below specifies an operation and an address of a word in memory. Associated with each instruction is a short mnemonic code called an *operation code*, or simply *code*, which is an abbreviation for the operation. Thus MOV is the code used for the load register instruction. Addresses are designated by such symbols as X, MAX, and HEAD. Each such address is actually a number that is used to form a memory address.

INSTRUCTION	CODE	EXAMPLE	ACTION
Load register	MOV	MOV X,AC	AC ← MEMORY[X+1;X]. Load the contents of the word whose address is X into register AC. The contents of memory are not altered.
Store register	MOV	MOV AC,HEAD	MEMORY[HEAD+1;HEAD] ← AC. Store the contents of register AC in memory word HEAD. Register AC is not altered.
Add to register	ADD	ADD MAX,AC	AC ← AC+MEMORY[MAX+1; MAX]. Add the contents of memory word MAX to the contents of register AC. Load the result into register AC. The contents of memory word MAX are not altered.
Subtract from register	SUB	SUB Y,AC	AC ← AC−MEMORY[Y+1;Y]. Similar to ADD.

The load instruction copies a designated word (a pair of adjacent bytes) of memory into register AC in the processing unit. The contents of memory are unchanged by this instruction. The store instruction copies the word in the AC into a designated word of memory. The contents of the register referenced by the store instruction are left unchanged by the execution of the instruction. Note that the same operation code is used for both the load and store register instruction. The distinction between the two operations arises from the order of the addresses. The first address is considered to be the source of the data; the second is the destination. Hence for a load the AC is the destination of the data, whereas for a store instruction the AC is the source. The add instruction differs from the load instruction in that a datum is not merely copied into a register, but is added to the current contents of a register.

The column labeled ACTION is intended to show the effect of the execution of the instruction. However, the execution is actually more intricate than indicated there. Recall that every load and store operation

must set an address in the MAR and must transfer data to or from the MDR. A more detailed description of the MOV X,AC instruction is the following:

MAR ← X	Load address X into register MAR.
READ	Set READ/WRITE control to READ.
WORD	Set BYTE/WORD control to WORD.
GO	Set GO/IDLE.
MDR ← MEMORY[. . . ;MAR]	The contents of memory word specified by register MAR (word X) are placed in register MDR.
AC ← MDR	The contents of register MDR are transferred into register AC.

The instructions above are unlike those for Turing machines, but all aspects of Turing machine instructions are present. Note that the memory address portion corresponds to the part of the Turing machine instruction that specifies head movement. Instead of using a sequence of head-movement instructions to find a datum, as we might do with a Turing machine, we can access the datum in one operation by specifying its address in an instruction.

The portion of a Turing machine instruction that specifies the symbol to be written on the tape corresponds roughly to the operation code of the instruction. The load instruction, for example, copies data from memory to register AC. The data in memory may be viewed as symbols in remote areas of a tape that are rewritten as they are read. The register AC may be viewed as a portion of a tape whose previous contents are erased as a copy of the remote data is written into it.

There is one other aspect of Turing machine instructions that bears discussion. Recall that each such instruction specifies the next state of the machine. As mentioned earlier, the P register contains a number that corresponds to the machine state. As we might expect, the digital computer instructions specify how to modify the contents of the P register to change the state of the processing unit. For a full understanding of how this is done we must consider the notion of the stored program.

2.3 THE STORED PROGRAM

Up to this point we have discussed instructions for computers without considering how they are stored in the computer. The computer memory provides an ideal place to store them, since, when a program has run to completion, the memory can be erased and replaced with a new set of instructions. Moreover, random-access addressing of memory guarantees that the successor of each instruction can be retrieved in one memory cycle regardless of its location in memory.

The P register in the processing unit contains the address of the instruction undergoing execution. Instructions occupy two, four, or six bytes of memory. At the completion of an instruction the P register has been modified to indicate the address of the next instruction. Most instructions alter the P register by adding their length in bytes to the P register. Thus the address in P at the close of execution of one instruction is the address of the immediately following instruction. For example, consider the sequence of the following two instructions.

ADDRESS	LABEL	CODE	OPERAND	COMMENTS
100		MOV	ZERO,AC	Move MEMORY[. . . ;ZERO] to AC. Increase P by 4, the length of this instruction.
104		ADD	ONE,AC	Add the contents of MEMORY[. . . ;ONE] to AC. Increase P by 4, the length of this instruction.

In this case the memory cells at addresses named ZERO and ONE contain the integers 0 and 1, respectively. The MOV instruction occupies four bytes and terminates with P increased by 4. Hence the instruction executed immediately following MOV is taken from address 104, and thus the execution of a program proceeds sequentially from one instruction to the immediately adjacent instruction in memory. At the completion of the execution of the ADD instruction, the accumulator contains the value 1.

Not every instruction updates the P register as indicated in the above example. It is necessary to have instruction execution proceed nonconsecutively in certain instances and to be able to control the sequencing of instructions as a function of values calculated during program execution. As a simple example of nonconsecutive sequencing of instructions, consider the program loop shown in this example:

ADDRESS	LABEL	CODE	OPERAND	COMMENTS
100		MOV	MINUS10,AC	Move the contents of MEMORY[. . . ;MINUS10] to AC. Increase P by 4.
104	X:	ADD	ONE,AC	Add the contents of MEMORY[. . . ;ONE] to AC. Increase P by 4.
108		BMI	X	Change P to 104 if last instruction left AC less than 0; otherwise, increase P by this instruction length.

In this example we assume that MEMORY[...;MINUS10] contains the number −10. The BMI instruction (branch on minus) causes the instruction at MEMORY[...;X] to be executed if AC contains a negative number; otherwise, it passes control to the instruction immediately succeeding the BMI instruction in memory. In this example, the MOV instruction initializes AC to the value −10, and the two instructions ADD and BMI are executed repeatedly with the value in the AC increasing by 1 each iteration until it reaches 0. At this point, when the BMI is executed, the P register is not changed to 104 but is increased to the address of the instruction following the BMI. This three-instruction sequence shows how to construct program loops. For example, the sequence of instructions in this example is executed exactly 10 times.

The action of a branch instruction can modify the P register to force the next instruction to be nonconsecutive. If, as in this example, the branch depends on conditions that occur during program execution, it is called a *conditional branch*. Otherwise, it is called an *unconditional branch*. The PDP-11 has several branch instructions, each useful for a particular program context. The instructions make use of special registers known as the *condition code* registers to determine how to change the P register. Arithmetic instructions such as the ADD cause the condition code registers to indicate whether the results of the arithmetic operation are negative, positive, or zero. The branch instructions test the condition code registers.

More specifically, two of the condition code registers are the Z (zero) and N (negative) registers. Each is one bit in length. A value of 1 in Z indicates a zero result; a value of 1 in N indicates a negative result. An example of some instructions that set the condition code is shown below.

INSTRUCTION	CODE	EXAMPLE	ACTION
Add to register	ADD	ADD MAX,AC	AC ← AC+ MEMORY[. . . ;MAX]; **if** result < 0 **then** N ← 1 **else** N ← 0; **if** result = 0 **then** Z ← 1 **else** Z ← 0;
Subtract from register	SUB	SUB L,AC	AC ← AC−MEMORY[. . . ;L]; **if** result < 0 **then** N ← 1 **else** N ← 0; **if** result = 0 **then** Z ← 1 **else** Z ← 0;

INSTRUCTION	CODE	EXAMPLE	ACTION
Increment register	INC	INC AC	AC ← AC+1; **if** AC < 0 **then** N ← 1 **else** N ← 0; **if** AC = 0 **then** Z ← 1 **else** Z ← 0;
Decrement register	DEC	DEC AC	AC ← AC−1; **If** AC < 0 **then** N ← 1 **else** N ← 0; **if** AC = 0 **then** Z ← 1 **else** Z ← 0;

Five of the PDP-11 branch instructions appear below. Note that a branch instruction is one word long.

INSTRUCTION	CODE	EXAMPLE	ACTION
Branch unconditional	BR	BR X	P ← X. An unconditional branch occurs to the instruction beginning at MEMORY[. . . ;X]. The condition codes do not affect this instruction.
Branch on plus	BPL	BPL MAX	If N = 0 then P ← MAX; otherwise increment P to the address of the following instruction (P ← P+2).
Branch on minus	BMI	BMI LOOP	If N = 1 then P ← LOOP; otherwise P ← P+2.
Branch on equal	BEQ	BEQ MATCH	If Z = 1 then P ← MATCH; otherwise P ← P+2.
Branch on not equal	BNE	BNE NEXT	If Z = 0 then P ← NEXT; otherwise P ← P+2.

A listing of the condition code settings for all the instructions in a given model of the PDP-11 appears in the manual entitled *PDP-11 Processor Handbook*, available from Digital Equipment Corporation. Although there are several different models of the PDP-11, this text treats those aspects of the PDP-11 common to all models. The condition code settings of other instructions used in examples to control conditional branches are introduced later as required. As a rule of thumb,

however, the obvious meaning of negative (N) and zero (Z) with regard to the given operation can be used in almost all cases.

The following program illustrates the BPL and SUB instructions. The instruction labeled LOOP is executed 11 times.

ADDRESS	CODE	OPERAND	COMMENTS
	MOV	ZERO,AC	Initialize AC to 0.
LOOP:	SUB	TEN,AC	Are contents of AC $\geq$ 10?
	BPL	NEXT	Yes, continue with other processing.
	ADD	ELEVEN,AC	No, restore AC to value before subtraction and increase by 1.
	BR	LOOP	Repeat, starting at LOOP.
NEXT:			

The operands identified as ZERO, TEN, and ELEVEN are the addresses of memory words that contain the decimal numbers 0, 10, and 11, respectively.

Another way to control repetitive portions of a program is to use the INC instruction. The instruction labeled LOOP in the example below is repeated 11 times. The memory cell identified as MINUS11 contains the decimal number -11.

ADDRESS	CODE	OPERAND	COMMENTS
	MOV	MINUS11,AC	Initialize AC to -11.
LOOP:	INC	AC	Increment AC.
	BMI	LOOP	Branch to instruction at address LOOP if condition code N = 1.
NEXT:	...		

In order to store instructions in memory we must be able to encode them into combinations of bits. Similarly, we must encode numbers as combinations of bits. Moreover, since the contents of a memory cell are simply a collection of bits, there must be a mechanism that decides whether the bits are to be interpreted as a number or as an instruction, because a memory cell may contain either entity. The distinction between positive and negative numbers also must be made. The next three sections will treat the representation of positive, negative, and floating-point numbers in memory. The representation of instructions will be postponed until Section 2.7.

2.4 NUMBER REPRESENTATIONS BASED ON A BINARY RADIX

The decimal notation 1234 is interpreted as a number that is the sum of one 1000, two 100's, three 10's, and four 1's, or, more succinctly,

$$1234 = 1 \cdot 10^3 + 2 \cdot 10^2 + 3 \cdot 10^1 + 4 \cdot 10^0$$

The abstract notion of *number* is quite independent of decimal representation. It is a matter of convention that we represent the numbers by a sequence of decimal symbols. Because of the inherent binary characteristic of computer components, numbers in computer systems are represented internally by sequences of binary symbols rather than decimal symbols. Both binary representation and decimal representation of numbers are examples of *radix representations*.

Given any positive integer b, where $b \geq 2$, the radix-b representation of a positive integer N is the sequence of digits $n_{m-1}, \ldots, n_1, n_0$, where $0 \leq n_i \leq b - 1$, $N < b^m$, and $N = \Sigma n_i b^i$. It is a relatively straightforward exercise in algebra to show that the radix-b representation of a positive integer N is unique, except possibly for leading 0's. Hence 1234 is the unique decimal representation of our example number, except that an arbitrary number of 0's may be prefixed to the representation. We shall indicate the radix of a representation by a subscript, as in 1234_{10}, when it is not clear from context.

Decimal arithmetic generalizes to other radices. For example, radix-b addition is similar to decimal addition, except that carries are generated whenever a sum digit is b or larger instead of 10 or larger. For example, the usual arithmetic operations are as follows in the radices indicated.

$$\begin{array}{r} 34112_5 \\ +24441_5 \\ \hline 114103_5 \end{array} \qquad \begin{array}{r} 75314_9 \\ -66525_9 \\ \hline 7678_9 \end{array} \qquad \begin{array}{r} 21332_4 \\ \times\ 133_4 \\ \hline 131322 \\ 131322 \\ 21332 \\ \hline 10311002_4 \end{array} \qquad \begin{array}{r} 12_3 \\ 212_3\overline{)11021_3} \\ 212 \\ \hline 1201 \\ 1201 \\ \hline \end{array}$$

Although binary representation is basic to computers, for practical convenience radix-8 and radix-16 representations are used as shorthand notation in the computer world. They are easily derived from binary representation by partitioning binary symbols into groups of three or four and finding their base-8 or base-16 equivalents. For example,

$$1010011101000_2 = 101\ 001\ 110\ 100_2$$
$$= 5\quad 1\quad 6\quad 4_8$$

The letters A to F are used to represent integers 10 to 15 in radix-16 representation. Hence

$$1010\ 0111\ 0100_2 = A74_{16}$$

Representation of fractions in nondecimal radices is a natural generalization of decimal fraction representation. To the right of a binary point, for example, the symbols in a binary fraction representation are the coefficients of 2^{-1}, 2^{-2}, etc. Thus 101.1101 is the binary representation of 5.8125_{10}, computed as

$$\begin{aligned}
1 \cdot 2^2 &= 4 \\
0 \cdot 2^1 &= 0 \\
1 \cdot 2^0 &= 1 \\
1 \cdot 2^{-1} &= .5000 \\
1 \cdot 2^{-2} &= .2500 \\
0 \cdot 2^{-3} &= 0 \\
1 \cdot 2^{-4} &= .0625 \\
&\overline{5.8125_{10}}
\end{aligned}$$

2.5 REPRESENTATION OF NEGATIVE NUMBERS

Representation of negative numbers has not been standardized in the computer industry. Four alternative schemes are discussed in this chapter. Their relative advantages and disadvantages are related mainly to the cost and speed of operation of the arithmetic processor. Since this aspect is beyond our present scope, no attempt is made to evaluate the various schemes. The computer specialist is likely to come in contact with all four representations in the course of his career.

2.5.1 Two's-Complement Representation

We use the notation $\bar{X}$ to mean the *complement* of X, where X is a vector of 0's and 1's. To obtain $\bar{X}$ from X, we change the 0's in X to 1's and the 1's to 0's. The complement operation is of central importance in the representation of negative numbers.

Let X be the binary representation of a number N. Then the number $-N$ in two's-complement representation is defined to be $\bar{X} + 1$. By

assumption in our discussion of the various methods for representing negative numbers, the representation of a number has exactly m binary bits, where m is an integer constant and is equal to the number of bits per memory word. We have to be careful to consider what happens if adding the integer 1 to $\bar{X}$ causes the representation to exceed m bits. The two's complement of X is so defined that carries that leave the most significant bit position are lost. For example, the two's complement of $11\ldots1_2 + 00\ldots01_2$ is $00\ldots0_2$. When we do the addition, we obtain a carry out of the leftmost bit position; this is disregarded, yielding the result $00\ldots0$. Since the number whose representation is $00\ldots0_2$ is the number 0_{10}, we see that the negative of 0_{10} is 0_{10}, as it should be. To distinguish between positive and negative numbers, it is usual practice to use the leading bit of the binary representation for a sign bit. If the sign bit is 0, the number is positive; if the sign bit is 1, the number is negative.

What is the range of integers that can be represented in two's-complement form when m is the number of bits per word? To answer this, note that positive integers can be represented by setting the first bit to 0 and using the remaining $m - 1$ bits to represent the integer magnitude. All integers in the range $0 \leq x \leq 2^{m-1} - 1$ can be represented as positive integers. For negative integers the sign bit is 1, and the remaining $m - 1$ bits are available to represent the magnitude of the integer. The representation of -1 is $111\ldots1$, and the representation of the most negative integer is $1000\ldots0$, which is the representation of -2^{m-1}. Hence all numbers in the range $-2^{m-1} \leq x < 2^{m-1}$ are representable. In the PDP-11, for example, $m = 16$ and numbers in the range $-2^{15} = -32{,}768 \leq x \leq 2^{15} - 1 = -32{,}767$ are representable. Addition of two numbers in this representation system follows two rules:

1. First add the two numbers and observe if a carry occurs into the sign bit and if a carry occurs out of the sign bit.
2. If no carries occur or if both carries occur, the sum is in range. If one, and only one, of the two carries occurs, there has been an overflow.

Example 2.1 shows that arithmetic in two's-complement representation leads to consistent results. That is, when we add the representations of two numbers, we obtain the representation of their sum, provided that no overflow occurs. It is easy to see why this is so. Let the radix representation of the positive integer N be $n_{m-1}, \ldots, n_1, n_0$, so that $N = \Sigma n_i 2^i$. We say that the digit n_i has a weight of 2^i, since it is the coefficient of 2^i in the summation. Two's-complement representation is a representation in which the sign digit n_{m-1} has a *negative weight* of

EXAMPLE 2.1

The following arithmetic operations use a word length $m = 4$:

0101	5	Carry into sign bit, overflow.
0110	6	
1011	11	
0101	5	
0010	2	
0111	7	
0101	5	
1010	−6	
1111	−1	
0110	6	Carry into sign and out of sign bit, no overflow.
1011	−5	
0001	1	
1011	−5	Two carries, no overflow.
1111	−1	
1010	−6	
1011	−5	Overflow, true result is negative because of carry out of
1010	−6	sign position.
0101	−11	

-2^{m-1} instead of a positive weight of 2^{m-1} (see Exercise 2.9). Since arithmetic with ordinary radix representations is consistent, two's-complement arithmetic is consistent, except possibly for arithmetic operations that generate carries into and out of the sign position. The overflow rules specify the conditions for correct results in these cases.

To clarify the overflow rules, let us consider what happens when we add two positive integers. The sum of two positive integers must be positive; and to be representable, this sum cannot exceed $2^{m-1} - 1$. If it should exceed $2^{m-1} - 1$, a carry into the sign position occurs without an accompanying carry out of the sign position. Hence this particular carry condition is an overflow condition. It is easy to check that the sum of a positive and a negative number can never overflow, and such a sum always produces either zero or both carries that involve the sign position. The last case to consider is the sum of two negative numbers. When does such a sum overflow, and how is the overflow condition distinguished by the carries?

In the PDP-11 computer, if an overflow is sensed by the processor, according to rule 2 above, a condition code register V similar to N and Z, which are discussed in Section 2.3, is set to 1. The state of the overflow

condition code V can be sensed by the instruction branch on overflow set (BVS), which causes the computer to branch to the designated address if V is 1. The instruction branch on overflow clear (BVC) causes a branch to the designated instruction if V is 0. A fourth condition code register is the C (carry) register. The C register is set or cleared depending on whether the last operation caused a carry from the most significant bit. The branch on carry set (BCS) and branch on carry clear (BCC) instructions sense the state of the carry condition code C. The effect of these four branch instructions that sense the condition code registers V and C is summarized as follows:

INSTRUCTION	CODE	EXAMPLE	ACTION
Branch on overflow set	BVS	BVS ERROR	**if** V = 1 **then** P ← ERROR **else** P ← P+2;
Branch on overflow clear	BVC	BVC NEXT	**if** V = 0 **then** P ← NEXT **else** P ← P+2;
Branch on carry set	BCS	BCS AD	**if** C = 1 **then** P ← AD **else** P ← P+2;
Branch on carry clear	BCC	BCC CONT	**if** C = 0 **then** P ← CONT **else** P ← P+2;

An example of the effect of some instructions on all four condition codes is shown below. The compare (mnemonic CMP) instruction subtracts the destination from the source by using two's-complement arithmetic and sets the condition codes according to the results. The source and destination operands are not altered. Note that not all instructions affect all the condition codes. For example, the increment (INC) instruction does not alter the C condition code.

INSTRUCTION	CODE	EXAMPLE	ACTION
Add to register	ADD	ADD MAX,AC	AC ← AC+ MEMORY[. . . ;MAX]; **if** result < 0 **then** N ← 1 **else** N ← 0; **if** result = 0 **then** Z ← 1 **else** Z ← 0; **If** carry **then** C ← 1 **else** C ← 0;

INSTRUCTION	CODE	EXAMPLE	ACTION
			if overflow **then** V ← 1 **else** V ← 0;
Increment	INC	INC AC	AC ← AC+1; **if** AC < 0 **then** N ← 1 **else** N ← 0; **if** AC = 0 **then** Z ← 1 **else** Z ← 0; **if** overflow **then** V ← 1 **else** V ← 0;
Compare register	CMP	CMP AC,L	**if** AC—MEMORY[. . . ;L] < 0 **then** N ← 1 **else** N ← 0; **if** AC—MEMORY[. . . ;L] = 0 **then** Z ← 1 **else** Z ← 0; **if** carry **then** C ← 1 **else** C ← 0; **if** overflow **then** V ← 1 **else** V ← 0;

EXAMPLE 2.2

The following shows how to add two numbers and test for overflow:

```
MOV     X,AC    Load AC with X.
ADD     Y,AC    Add Y.
BVS     ERR     Branch to first instruction of overflow processing
                sequence.
BR      EXIT    Normal exit.
...
```

The conditional branch instructions that have been encountered so far have initiated a branch based only on the value of a single condition code. More complex branch conditions based on a combination of the condition codes are often useful. The PDP-11 has several complex conditional branch instructions as exemplified below.

INSTRUCTION	CODE	EXAMPLE	ACTION
Branch on greater or equal	BGE	BGE LOOP	**if** (N = 0 AND V = 0) **then** P ← LOOP **else** **if** (N = 1 AND V = 1) **then** P ← LOOP **else** P ← P+2;
Branch on greater than	BGT	BGT NEXT	**if** (Z = 0 AND N = 0 AND V = 0) **then** P ← NEXT **else** **if** (N = 1 AND V = 1) **then** P ← NEXT **else** P ← P+2;
Branch on less than or equal	BLE	BLE FLAG	**if** Z = 1 **then** P ← FLAG **else** **if** (N = 0 AND V = 1) **then** P ← FLAG **else** **if** (N = 1 AND V = 0) **then** P ← FLAG **else** P ← P+2;
Branch on less than	BLT	BLT ERR	**if** (N = 0 AND V = 1) **then** P ← ERR **else** **if** (N = 1 AND V = 0) **then** P ← ERR **else** P ← P+2;
Branch higher	BHI	BHI BIGGER	**if** C = 0 **then** P ← BIGGER **else** **if** Z = 0 **then** P ← BIGGER **else** P ← P+2;
Branch on lower or same	BLOS	BLOS LESS	**if** C = 1 **then** P ← LESS **else** **if** Z = 1 **then** P ← LESS **else** P ← P+2;

Most computers have instructions that subtract the contents of a specified memory location from a machine register. The PDP-11 has such an instruction, the SUB instruction. During execution of such an instruction the computer adds the negative of the subtrahend to the contents of the machine register. Some computers do not have such an instruction, so that the negative must be computed explicitly. Alter-

natively it may be necessary to form the negative of a number. For these purposes we make use of instructions such as complement (code COM), negate (code NEG), and increment (code INC).

INSTRUCTION	CODE	EXAMPLE	ACTION
Complement	COM	COM AC	AC ← $\overline{AC}$;
Negate	NEG	NEG AC	AC ← $\overline{AC}+1$;

These instructions can be used as follows for computing the negative of the contents of the AC.

```
COM     AC      Complement AC
INC     AC      Add 1
```

The above sequence of code can be shortened by simply using the instruction NEG AC.

Two's-complement representation is preferable to other representations for the PDP-11 computer because the programming is slightly easier, as we shall see later. Computers that have subtraction built in are based on a particular representation of negative numbers and usually do not support efficient computation with other representations.

2.5.2 Ones'-Complement Representation

In the ones'-complement system, if X is the binary representation of the integer N, then $-N$ is represented by $\bar{X}$. Since a 1 is not added to the complement of a number, as in two's-complement arithmetic, special action is necessary to ensure that computations will generate answers in the proper representation. With respect to addition, no action need be taken if two positive numbers are added; but if two negative numbers are added, the answer will be too small by 1. To see this, compare the addition of two negative numbers in both ones'-complement and two's-complement arithmetic. Let X and Y be the positive integers and calculate $(-X) + (-Y)$. Note that the results below show a numerical difference of 2 in the representations when there should be a difference of only 1.

$$\begin{array}{r} \overline{X}+1 \\ +\overline{Y}+1 \\ \hline \overline{X}+\overline{Y}+2 \end{array}$$ Two's complement

$$\begin{array}{r} \overline{X} \\ +\overline{Y} \\ \hline \overline{X}+\overline{Y} \end{array}$$ Ones' complement

The following rules for addition state precisely how to determine when 1 must be added to the sum and how to determine whether overflow occurred:

1. Add the two operands.
2. If a carry out of the sign occurs, then add 1 to the sum (this is called an *end-around carry*).
3. Inspect carries into the sign and out of the sign. If only one of the two carries has occurred, then overflow has occurred.

EXAMPLE 2.3

The following examples show ones'-complement addition for operands of length $m = 4$:

0101	5	Carry into sign bit. Overflow.
0110	6	
1011	11	
0101	5	
0010	2	
0111	7	
0101	5	
1001	−6	
1110	−1	
0110	6	Two carries occur; end-around carry added to intermediate sum.
1010	−5	
0000	1	
+1		
0001		
1101	−2	Two carries occur; end-around carry added to intermediate sum.
1100	−3	
1001	−5	
+1		
1010		
1101	−2	End-around carry, but no carry into sign bit. Overflow.
1001	−6	
0110	−8	
+1		
0111		

Following the reasoning of the previous section, it is a relatively straightforward matter to establish the correctness of the rules for ones'-complement addition and to prove that the representation system is consistent. In particular, ones'-complement representation is a representation in which the weight of the sign bit is $-(2^{m-1} - 1)$.

Ones'-complement arithmetic operations can be programmed for the PDP-11 computer by implementing rule 2. The computers that have ones'-complement representation built into their arithmetic hardware automatically follow rule 2, thereby relieving the programmer of a minor burden.

EXAMPLE 2.4

The following program illustrates how to implement rules 2 and 3 on the PDP-11 computer. If a carry out of the sign position occurs, then condition code C is set to 1. The instruction clear (code CLR) places a 0 in the designated location. Individual condition codes can be set or cleared by such instructions as clear overflow register V (code CLV).

	LABEL	CODE	OPERAND	COMMENTS
1.		CLR	AC	Clear AC to count carries.
2.		MOV	AC,COUNT	Store zero into MEMORY[. . . ;COUNT].
3.		MOV	X,AC	First operand to register AC.
4.		ADD	Y,AC	Add second operand.
5.		BVC	NEXT1	Branch to NEXT1 if 0 or both carries occurred.
6.		INC	COUNT	Increment MEMORY[. . . ;COUNT] if one carry occurred.
7.		CLV		Clear overflow, prepare for end-around carry.
8.	NEXT1:	BCC	NEXT2	Check for end-around carry. Branch if no end-around carry.
9.		INC	AC	Increment AC if carry out of sign occurred.
10.		BVC	NEXT2	Check for overflow on the preceding instruction, branch to NEXT2 if no overflow.

	LABEL	CODE	OPERAND	COMMENTS
11.		INC	COUNT	Increase MEMORY[. . . ;COUNT] to indicate another carry.
12.	NEXT2:	MOV	AC,X	Save the sum.
13.		MOV	COUNT,AC	Load AC with MEMORY[. . . ;COUNT].
14.		CMP	AC,ONE	Check for odd number of carries.
15.		BEQ	OFLOW	Branch to overflow program if exactly one carry occurred.
16.				

The instruction sequence can be shortened somewhat if overflow cannot occur or is not important. The following instruction sequence computes $X - Y$ without overflow checking.

```
        MOV     Y,AC
        COM     AC      Ones' complement of Y in AC.
        ADD     X,AC    X — Y is in AC.
        BCC     NEXT    Branch if carry zero.
        INC     AC      Add end-around carry.
NEXT2:  ...
```

Note that this instruction sequence is three instructions longer than a two's-complement subtraction with a SUB instruction.

The range of representable numbers in the ones'-complement system is of interest because it is slightly different from the range for two's-complement representation. Again, we let the first bit indicate the sign of a number and use the remaining $m - 1$ bits of an m-bit computer word to represent the magnitude of the number. The range of positive representable integers must be the same for both ones'- and two's-complement representations, since the representation of positive integers is the same in both systems. The most negative representable number in the ones'-complement system must be 1 greater than the most negative two's-complement number, because negative representations in the two systems differ by 1. Hence the range of representable numbers in the ones'-complement system is $-2^{m-1} + 1 \leq N \leq 2^{m-1} - 1$. For the PDP-11, where $n = 16$, this range is $-32{,}767 \leq x \leq 32{,}767$. Note that the range is symmetric with respect to zero, whereas for two's-complement representation it is not. The significance of the symmetry is that, in the ones'-complement system, if N is representable, then so is $-N$. However, in the two's-complement system there is a representable inte-

ger N such that $-N$ is not representable. Which integer is this? What happens when we attempt to negate this integer with the NEG instruction?

There is an interesting anomaly in the ones'-complement system that can cause the programmer some frustration. There are two different representations of zero in this system, since both $00\ldots0_2$ and $11\ldots1_2$ satisfy the usual rules that we reserve for zero. To differentiate between the two zeros we usually refer to $00\ldots0_2$ as *plus zero* and to $11\ldots1_2$ as *minus zero*. To verify that minus zero is truly a zero consider the following examples:

```
 0101        5
 1111      +(−0)
-----      -----
10100        5
    1
-----
 0101        End-around carry

 0101        5
 1010       −5
-----      -----
 1111       −0
```

When the integer N is added to its negative, $-N$, in ones'-complement notation, the result is always minus zero rather than plus zero. When it is necessary to check to see if a result of a computation is equal to zero, the result must be compared to *both* plus zero and minus zero, whereas only a single comparison is necessary for two's-complement notation, since the representation of zero is unique.

2.5.3 Signed-Magnitude Representation

Each integer in this representation system is represented by m bits, the first of which is a *sign bit*. The remaining $m - 1$ bits represent the magnitude of the integer in the conventional binary system. If $m = 4$, then representations of $+5$ and -5 are 0101 and 1101, respectively. Arithmetic in this system can be programmed on the PDP-11 computer, but with considerably more difficulty than either of the other two representations. The machines that use this representation have hardware that does arithmetic operations with signed-magnitude operands and produces signed-magnitude results without burdening the programmer with the details of representation. The range of representable numbers is identical to that for ones'-complement representation, and both plus zero and minus zero are representable.

2.5.4 Excess-2^{m-1} Representation

In this number system the representation of the number N, where $-2^{m-1} \leq N \leq 2^{m-1} - 1$, is the ordinary binary representation of $N + 2^{m-1}$. Clearly $0 \leq N + 2^{m-1} \leq 2^m - 1$, so that $N + 2^{m-1}$ is guaranteed to be a positive integer that is representable in m bits. It is rather interesting that this number representation system is identical to the two's-complement system, except that the sign bits are coded differently. A sign of 0 indicates a positive integer in the two's-complement system, whereas it indicates a negative integer in the excess-2^{m-1} system. Thus $1100000 = 32_{10}$ in the excess-64 system, whereas $0100000_2 = 32_{10}$ in the two's-complement system.

The relation of the excess-2^{m-1} to the two's-complement system makes it particularly simple to derive the rules for detecting overflow. How can we use the overflow-sensing instructions to detect overflow in this system?

2.6 FLOATING-POINT REPRESENTATION

In the number representations discussed thus far the computer hardware is insensitive to the location of the binary point. For example, suppose 0101110_2 and 0001011_2 are two operands; their sum is 0111001_2. If the binary point were to the right of the rightmost bit, then the two operands would be 46_{10} and 11_{10}, respectively, and their sum would be the representation of 57_{10}. However, if the binary point were between the fourth and fifth bits, counting from left to right, the operands would be the representations of 5.750_{10} and 1.375_{10}, respectively, and their sum would be the representation of 7.125_{10}. This example suggests that the programmer may establish a binary point in any position that he chooses. Of course, he must assure that the binary points of operands are aligned for addition and subtraction operations. Moreover, the position of the binary point of quotients and products depends on the position of the binary points of their operands. Hence for multiplication and division operations the programmer must adjust the results of the operation to account for a possible discrepancy between the actual position of the binary point and its desired position.

It is possible to simplify the bookkeeping related to binary-point locations by adopting the binary equivalent of scientific notation. In the usual scientific notation a number is represented by a mantissa and a power of 10—for example, $1.234 \cdot 10^7$. The exponent of 10 merely designates the true location of the decimal point, and the mantissa 1.234 contains the significant digits of the number. The decimal point is placed in a fixed position, usually to the right of the first digit. The representations

that we have discussed thus far encode only the mantissas of numbers, and not the binary points. The location of binary points is implicit in the computer representation. Consequently, by appropriate scaling the programmer can fix the binary point in any location he desires. Number representations of data internal to computers in which the binary point is implicit are called *fixed-point representations*. Representations in which the location of the binary point is encoded separately from the mantissa are called *floating-point representations*. Scientific notation is an example of a floating-point representation.

A great many different kinds of floating-point representations are possible; in fact, at least a dozen different ones are in use on commercial computers. A typical representation is as follows:

Sign	Sign of exponent	Exponent	Mantissa
0	0	001010	1000000000

In this example we assume the binary point of the mantissa is between the first and second bits of the mantissa, and the exponent is assumed to be an integer. Thus the example represents the number $1.0 \cdot 2^{10} = 1024_{10}$. A positive exponent indicates that the binary point should be shifted to the right for a true representation, and a negative exponent indicates that a shift to the left is necessary. This convention is consistent with scientific notation.

The many variations possible for floating-point representations arise from the different possible representations for negative numbers and the possible locations for the implicit binary point in the mantissa. In yet another variation the exponent can be treated as an exponent of 8 or 16 instead of 2. In this case it is convenient to view the mantissa representation as a radix-8 or radix-16 representation in which the exponent describes the location of the octal point or the hexadecimal point, respectively. Most computers order the elements of the floating-point representation as shown above, although there are a few cases in which the exponent is located to the right of the mantissa.

Among the many representations that have been implemented, ones'-complement, two's-complement, signed-magnitude, and excess-2^{m-1} have all been used for the exponent. Similarly, the mantissa has been represented in ones'-complement, two's-complement, and signed-magnitude schemes. The sign representation of the mantissa generally follows the usual practice of 0 for positive and 1 for negative. Radix points of mantissas are usually assumed to be just in front of the leading digit, after the rightmost digit, or between the first and second digits, where the digits are binary, octal, or hexadecimal.

The three examples in Table 2.1 illustrate the wide variety of repre-

TABLE 2.1

	PDP-11 (TWO COMPUTER WORDS)	IBM SYSTEM/ 360-370	BURROUGHS B-5500 (BIT 0 NOT USED)
Exponent			
Sign position	1	1	2
Position	2 to 8	2 to 7	3 to 8
Representation	Excess 128	Excess 64	Signed magnitude
Mantissa			
Sign position	0	0	1
Position	9 to 31	8 to 31	9 to 47
Representation	Signed magnitude	Signed magnitude	Signed magnitude
Radix	2	16	8
Radix point	Before first digit	Before first digit	After last digit

sentations. For purposes of comparison we assume that the bits in a computer word are numbered from left to right, starting at 0.

In any floating-point representation there are numbers that do not have a unique representation. This characteristic is due to representations that have different exponents with appropriately scaled mantissas. Thus in scientific notation the numbers $1.00 \cdot 10^0$, $0.10 \cdot 10^1$, and $0.01 \cdot 10^2$ are all the same number, although they differ in the number of significant digits that follow the leading nonzero digit. In computers the usual practice is to choose a representation such that the leading digit of the mantissa is nonzero if the number is nonzero. This is called *normalized representation*. In radix-8 and radix-16 representations the leading octal digit or hexadecimal digit is nonzero but the leading bit may be 0. Special hardware in some computers automatically produces results of floating-point operations in normalized form.

The PDP-11 takes advantage of the fact that in normalized representation with radix 2 the leading digit is always 1. Thus this bit need never be represented. The leading 1 bit is called the *hidden bit*, and consequently the mantissa is effectively 24 bits. An exception to the hidden bit being 1 is the number zero. In order to represent zero, the PDP-11 adopts the convention that the hidden bit is 0 if the exponent is all zeros (the most negative exponent). Hence a floating-point zero and an integer zero have the same representation in the PDP-11.

In spite of the variety of representations, the following rules for converting decimal numbers to floating-point numbers can be applied generally.

1. Convert the decimal number to binary in the system of representation used for the mantissa.
2. Compute the number of digits that the radix point must be shifted in order to obtain a normalized representation; this is the exponent.
3. Convert the exponent into the representation used for exponents.
4. Combine the exponent and the normalized mantissa.

EXAMPLE 2.5

Represent $1/32$ in the IBM System/360 format.

1. $(1/32)_{10} = .00001_2$.
2. The representation is normalized by shifting the hexadecimal point one digit (four bits) right. Hence the exponent is -1.
3. The excess-64 representation of -1 is 0111111_2.
4. The full representation is

   0 0111111 1000 0000 0000 0000 0000 0000.

EXAMPLE 2.6

Represent $1/32$ in the PDP-11 format.

1. $(1/32)_{10} = .00001_2$.
2. The representation is normalized by shifting the binary point 4 digits right. Hence the exponent is -4.
3. The excess-128 representation of -4 is 01111100.
4. The full representation after eliminating the first bit of the mantissa is

   0 01111100 00 000 000 000 000 000 000 000.

2.7 REPRESENTATION OF INSTRUCTIONS IN COMPUTER MEMORY

We have discussed various ways of representing numbers in a computer memory, but we have not yet discussed how to represent instructions. It is essential that instructions be encoded into sequences of bits, and design considerations suggest that they be encoded to occupy whole words or multiples thereof. In the PDP-11 computer, for example, each word has two bytes or sixteen bits, so that each memory fetch brings a group of sixteen 0's and 1's from memory to the MDR. Hence it is convenient to encode instructions with lengths that are multiples of 16 bits.

The pair of 16-bit words shown in Figure 2.4 illustrates a typical pattern that might be fetched from two adjacent words in the memory of a PDP-11 computer. The bits are numbered 0 to 15, from right to left, following the manufacturer's convention. (This convention is exactly reversed for the IBM System/360 and System/370 computers.) Suppose the computer reaches the point at which it has to fetch the next instruction from memory. It initiates a READ operation to the word whose address is in the P register, and we assume the first word shown in Figure 2.4 appears in the MDR at the end of the memory cycle.

The computer interprets the pattern in the register in the following way. The four-bit segment of the word in bit 15 through bit 12 is interpreted as the operation code according to the encoding shown in Table 2.2. Table 2.2 is incomplete: not all possible operation codes are included; only the codes for instructions introduced so far are shown. Note that two operation codes are reserved to define another class of instructions, the single-operand class. The use of such *reserved operation codes* is very common in small computers. The 6-bit word segment from bit 11 to bit 6 is used to determine the address of the source operand, and bits 5 through 0 are used to determine the address of the destination. In this example the bit pattern 110111 for the source indicates the source is a memory location whose address is derived from the next memory word, called the

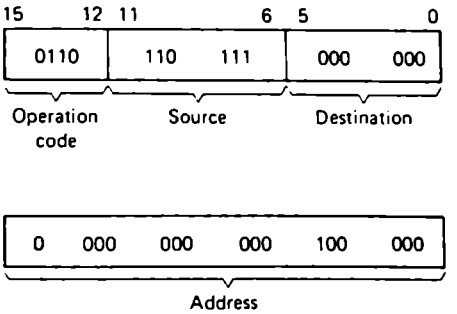


FIGURE 2.4 Double-operand instruction format for the PDP-11 computer

TABLE 2.2

PATTERN	CODE	
0001	MOV	
0010	CMP	
0110	ADD	
1110	SUB	
0000 1000		Specifies single-operand instruction

address word. The address is formed by adding the value in the address word of the instruction to the address of the word following the address word. In Figure 2.4, if the address word is at location 102_8, then the address of the source operand is $40_8 + 102_8 + 2_8 = 144_8$. The bit pattern for the destination 000 000 means that the accumulator (AC) is the destination. Thus the symbolic form for the bit pattern in Figure 2.4 is

```
ADD X,AC
```

The PDP-11 also has single-operand instructions. The format is shown in Figure 2.5. Since only one operand is specified, there is only one address field, which specifies both source and destination. Note that if the AC is specified as the destination (bits 5 through 0 are 000 000), then the second word is unnecessary and is omitted from the encoding. Bits 15 through 12 are either 0000 or 1000, and they define the single-operand class of instructions. Bits 11 through 6 are interpreted as a secondary-operation code according to Table 2.3. Again Table 2.3 is incomplete.

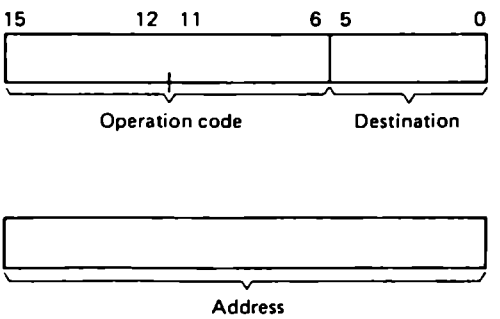


FIGURE 2.5 Single-operand instruction format for the PDP-11 computer

TABLE 2.3

PATTERN	CODE	
0000 000 001	JMP	
0000 101 000	CLR	
0000 101 001	COM	
0000 101 010	INC	
0000 101 011	DEC	
0000 101 100	NEG	
0000 0XX XXX 1000 0XX XXX		Specifies a branch instruction

The representation for a branch instruction is shown in Figure 2.6. Note that bits 15 through 11 must be either 00000_2 or 10000_2. Bits 10 through 8 are interpreted to yield the specified branch instruction according to Table 2.4. The offset of the branch instruction shown in Figure 2.6 is a two's-complement number that specifies the number of memory words the branch destination is displaced from the instruction following the branch instruction. Note that a branch instruction can only transfer control $+127$ or -128 words from the instruction following the branch instruction. Branching to an arbitrary instruction can be accomplished by the JMP instruction. The JMP instruction is a single-operand instruction that causes an unconditional branch to the instruction specified as the destination operand.

An essential aspect of the instruction representation is that there is no inherent difference in the physical storage of instructions and data. When the computer fetches a datum from memory, the word obtained is interpreted as a number. When the computer fetches an instruction, the word obtained is interpreted as an instruction. Therefore, two distinct memory cycles occur during execution of the ADD instruction and similar instructions. The first memory cycle obtains the instruction; the second memory cycle accesses an operand of the ADD.

In order to permit words fetched from memory to be interpreted in more than one way, the processor can assume one of four states during the execution of an instruction. This state determines how a word fetched

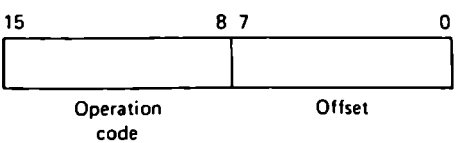


FIGURE 2.6 Branch instruction format for the PDP-11 computer

TABLE 2.4

PATTERN	CODE	ACTION
0 0000 001	BR	Branch
0 0000 010	BNE	Branch not equal (zero)
0 0000 011	BEQ	Branch on equal (zero)
0 0000 100	BGE	Branch greater or equal (zero)
0 0000 101	BLT	Branch less than (zero)
0 0000 110	BGT	Branch greater than (zero)
0 0000 111	BLE	Branch less than or equal (zero)
1 0000 000	BPL	Branch plus
1 0000 001	BMI	Branch minus
1 0000 010	BHI	Branch higher
1 0000 011	BLOS	Branch lower or same
1 0000 100	BVC	Branch overflow clear
1 0000 101	BVS	Branch overflow set
1 0000 110	BCC (BHIS)	Branch carry clear
1 0000 111	BCS (BLO)	Branch carry set

from memory is to be interpreted. The four states are FETCH, SOURCE, DESTINATION, and EXECUTE.

If a word is retrieved from memory while the processor is in the FETCH state, it is interpreted as an instruction. The word is placed in a processor register called the I register, which is shown in Figure 2.3. While the processor is still in the FETCH state, the instruction class is determined. If the instruction has a source operand, the operand is fetched in the SOURCE state and held in a register. Single- and double-operand instructions have the address of the destination operand calculated in the DESTINATION state. In this state the destination address is placed in the DEST register, where it is used during instruction execution.

When the processor completes its actions in the FETCH, SOURCE, and DESTINATION states, it switches to the EXECUTE state. In the EXECUTE state the contents of the I register are interpreted, and the instruction is obeyed. When, for example, the I register contains the ADD X,AC instruction, the SOURCE register, which holds the datum in MEMORY[. . . ;X], is added to the destination register AC and the result is placed in the AC rather than in the I register. While passing through these four states, the P register is always updated to point to the next consecutive memory word as each word it specifies is fetched from memory. On completion of

instruction execution the state is changed back to the FETCH state, and the processor repeats the actions for obtaining a new instruction.

To be more precise, the computer obeys the following description (the notation I_{x-y} denotes bits x to y of the I register; the address of the AC is 777 700):

```
FETCH state:        MAR ← P;
                    MDR ← MEMORY[...;MAR];
                    P ← P+2;
                    I ← MDR;
                    if I15–11 = 00000 OR I15–11 = 10000 then
                        begin comment branch instruction;
                            change state to EXECUTE
                        end
                    else
                    if I15–12 = 0000 OR I15–12 = 1000 then
                        begin comment single-operand instruction;
                            change state to DESTINATION
                        end
                    else
                        change state to SOURCE;

SOURCE state:       if I11–6 = 110 111 then
                        begin comment compute source address from
                        next memory word;
                            MAR ← P;
                            MDR ← MEMORY[...;MAR];
                            P ← P+2;
                            MAR ← MDR+P;
                            comment the source address is now in
                            MAR, now fetch the source operand;
                            MDR ← MEMORY[...;MAR];
                            SOURCE ← MDR;
                            change state to DESTINATION
                        end
                    else
                    if I11–16 = 000 000 then
                        begin comment source is AC;
                            SOURCE ← AC;
                            change state to DESTINATION
                        end;

DESTINATION state:  if I5–0 = 110 111 then
                        begin comment compute destination address
                        from next memory word;
                            MAR ← P;
                            MDR ← MEMORY[...;MAR];
                            P ← P+2;
                            DEST ← MDR+P;
                            change state to EXECUTE
                        end
```

```
                    else
                    if I5–0 = 000 000 then
                        begin comment AC is destination. Indicate
                        this as address 777 700₈;
                            DEST ← 777 700₈;
                            change state to EXECUTE
                        end;
EXECUTE state:      if I15–12 = 0110 then
                        begin comment ADD instruction execution;
                            MAR ← DEST;
                            MEMORY[...;MAR] ←
                              MEMORY[...;MAR] + SOURCE;
                            change state to FETCH
                        end
                    else
                    if I15–12 = 0001 then
                        begin comment MOV instruction execution;
                            MAR ← DEST;
                            MEMORY[...;MAR] ← SOURCE;
                            change state to FETCH
                    else
                      .
                      .
                      .
                      .
                    (other instructions)
```

In the DESTINATION state note that the AC register is given as memory address 777700_8. This memory address designates the AC register rather than a word in memory. No memory cycle is required to store results in the AC.

It is instructive to trace through the execution of the instruction given in Figure 2.4 by using the above description. Assume that the P register contains 100_8. The first word of the double-operand instruction is retrieved from memory and the P register incremented to 102_8. The state of the computer is changed to SOURCE, since the instruction is neither a branch nor a single-operand instruction. The bit pattern for the source operand is 110 111, which causes the word addressed by the P register (102_8) to be placed in the MDR. The P register is incremented to 104_8. The source-operand address is calculated by adding the second word of the instruction (now in the MDR) to the P register. Hence the source-operand address is $40_8 + 104_8 = 144_8$. The source operand is retrieved and placed in the SOURCE register. In the DESTINATION state the 000 000 bit pattern specifies the AC as the destination operand. The address of the AC is placed in the DEST register. Finally, in the EXECUTE state the operation code bit pattern 0110 for ADD causes the source

operand to be added to the destination operand. A more complete discussion of the addressing mechanism is presented in Chapter 4.

Since the instructions and data for a program reside in the same memory, it is possible to treat program instructions as data and execute a datum as if it were a program instruction. In fact, we might write a program that replaces an ADD X,AC instruction with an ADD Y,AC instruction or replaces a BR X with a BR Y instruction. Such a program is said to be *self-modifying*. For many years it was fashionable to write self-modifying programs, and indeed the design of early computers demanded that programs be self-modifying for the sake of efficiency. Although it has long been known that programs need not modify themselves, it has only recently been realized that programs that do not modify themselves enjoy certain advantages over self-modifying programs. In this text we shall adhere to the purist principle that programs never modify themselves.

REFERENCES

In 1945 Eckert and Mauchly constructed what is generally said to be the first digital computer. Their computer, ENIAC, had instructions similar to those described in this chapter, but programs were not stored in computer memory. Programming for ENIAC was done by wiring plug boards.

The concept of the stored program is generally credited to von Neumann, although Rosen (1969), in his historical survey of electronic computers, credits Eckert and Mauchly also. Burks, Goldstine, and von Neumann (1946) proposed plans for the first electronic digital computer at the Institute for Advanced Study. Their proposal is rather remarkable in its detail and sophistication and reflects nearly all the concepts of present-day computer operation. Moreover, two decades of advances in computer technology have not changed computers materially from the original proposal.

Construction of the IAS computer was begun in the late 1940s but not completed until 1952, by which time several other computers were also in operation. These early computers were quite similar to the computer described in this chapter; the major differences were attributable to the constraints on the memory organization imposed by the early memory technology. For a detailed treatment of the history of the development of computers see Bell and Newell (1971).

EXERCISES

2.1 Verify the following equalities:

$$67_{10} = 74_9 = 103_8 = 124_7 = 151_6 = 232_5 = 1003_4 = 2111_3$$

2.2 Convert 13441_{10} to its radix-16 equivalent. Convert 777_8 to radix 10 and $A3D2_{16}$ to radix 10.

2.3 State an algorithm for changing a number from radix-b representation to radix-d representation for any integers b and d greater than 1.

2.4 Prove that the radix-8 representation of any integer can be obtained from its radix-2 representation by partitioning the radix-2 representation into three-digit groups. (Note: Always partition from the binary point. $11011_2 = 11\ 011_2 = 33_8 \neq 110\ 11 = 63_8$.)

2.5 Find the binary representation of 5.3_{10}. Since 5.3_{10} can be represented exactly in radix-10 notation, why can it not be represented exactly in radix-2 representation? Find a number that cannot be represented exactly in radix-10 representation but can be represented exactly in some other radix.

2.6 Formulate algorithms for multiplication and division (*a*) when both operands are always positive and (*b*) when both operands are represented in two's-complement form. Assume that both operands are integers and that the dividend is a multiple of the divisor.

2.7 Formulate algorithms for addition and multiplication when the operands are in floating-point format.

2.8 Let the integer 1 be represented as a single 1 bit preceded by as many 0's as necessary to occupy a word of computer memory. If, by mistake, this number is treated as a floating-point number by a B-5500 computer, what is its value? What is its value in System/360-370 format? What is its value in PDP-11 format? (Note: The bits are assumed to be numbered from left to right in the formats given.) For the B-5500, System/360-370, and PDP-11 which integers have a floating-point representation identical to their signed-magnitude representation?

2.9 Prove that two's-complement representation is the same as the radix-2 representation of integers except that the sign bit has weight -2^{m-1}, where m is the number of bits per word. Prove that

the sign bit has weight $-(2^{m-1} - 1)$ in ones'-complement representation.

2.10 In ones'-complement addition what operands generate a carry into the sign bit during the addition of the end-around carry? (Note: There are several operand pairs that meet this criterion.)

2.11 The BGT (branch on greater than) instruction will initiate a branch if the condition codes N = 1 and V = 1. Is this consistent with the two's-complement representation of negative numbers? If so, why? If not, why not?

2.12 A datum in a memory word can be treated as a 16-bit unsigned integer. Assume a branch instruction is to be used after a compare instruction. For two's-complement and unsigned integers name the branch instructions that correspond to the result of the comparison being greater than zero, greater than or equal to zero, equal to zero, less than zero, and less than or equal to zero.

THREE

IBM SYSTEM/360-370 COMPUTER ORGANIZATION

The IBM System/360 and System/370 computers are comparable to the PDP-11 in several ways and differ in others that are significant to the programmer. In this chapter we shall consider the organization of these computers and a few examples of instructions for them.

3.1 MEMORY ORGANIZATION

Added flexibility is gained by being able to access data of varying length. In the PDP-11 computer each word is 16 bits. Moreover, all memory references cause an 8-bit byte or a 16-bit word to be transferred between the MDR and memory. It is impossible to access more than 16 bits with one instruction. The System/360-370 computers have a word length of 32 bits, but they can access 8, 16, 32, and 64 bits with one instruction.

A diagram of System/360-370 computer memory is shown in Figure 3.1*a*. Each addressable cell in memory is eight bits long and is called a byte. Groups of bytes, as well as individual bytes, can be accessed during the execution of a single instruction. For example, Figure 3.1*a* indicates that a pair of adjacent bytes, the first of which is even-numbered, can be accessed together as a group called a *halfword*. All the data lengths that can be accessed by a single instruction are as follows:

NAME	LENGTH	ADDRESS CONSTRAINT
Byte	8 bits	Any address.
Halfword	16 bits	First byte must have even address.
Full word	32 bits	First address must be divisible by 4.
Double word	64 bits	First address must be divisible by 8.

The constraints on addressing groups of bytes form natural boundaries within memory. The beginning of a byte with an even address is said to be a *halfword boundary* because a halfword may not cross this point. Full-word boundaries and double-word boundaries are defined similarly. Figure 3.1*b* shows the System/360-370 memory reorganized into words. Note that the addresses of the bytes that form a word are in the reverse order from the PDP-11 convention. That is, the byte with the smallest address is the leftmost byte in the word.

3.2 ORGANIZATION OF THE PROCESSING UNIT

The processing unit of System/360-370 computers has substantially more registers than the PDP-11 computer. There are two different sets of registers, one for floating-point arithmetic operations and the other for general-purpose operations. These are shown in Figure 3.2. There are four floating-point registers, each 64 bits in length. The usual convention for identifying registers is to use the integers 0, 2, 4, and 6. Figure 3.2 indicates that the four floating-point registers are used as four double-word or four full-word registers. In the latter case only the left half of

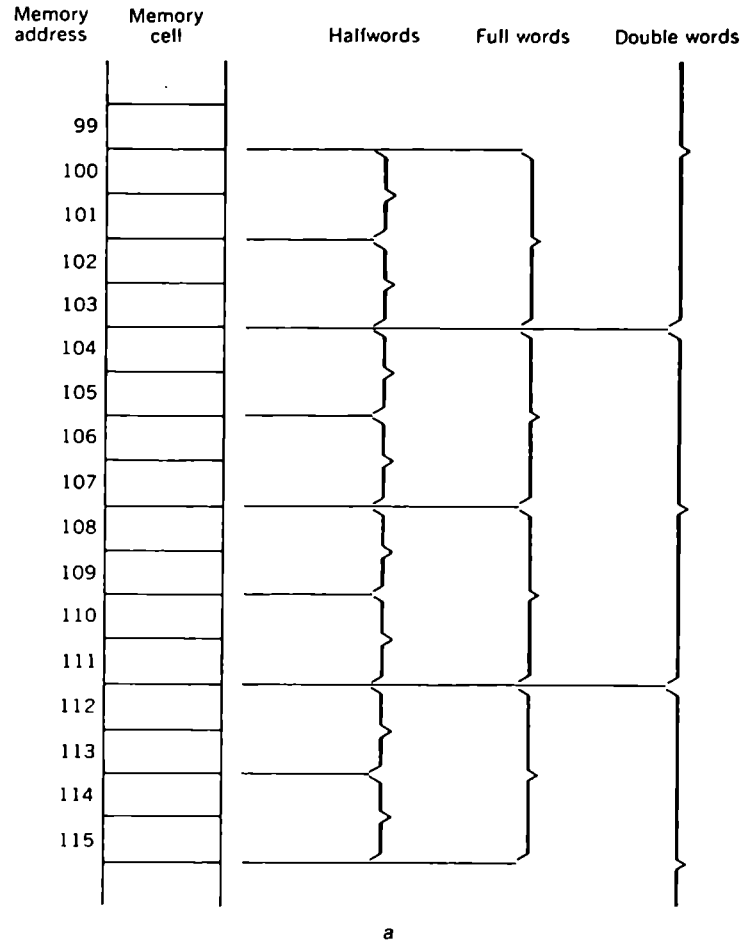


FIGURE 3.1 (*a*) The organization of System/360-370 memory; (*b*) the organization of the System/360-370 memory into words

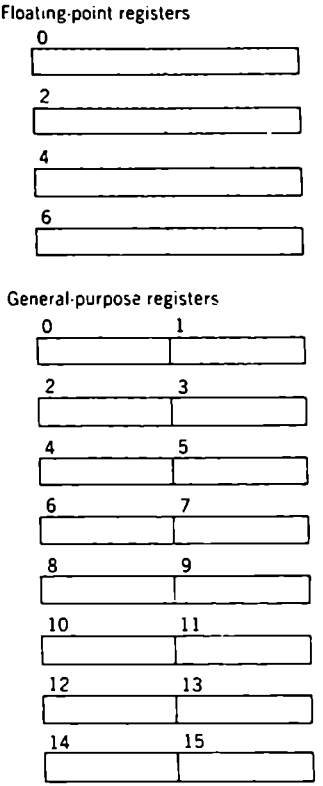


FIGURE 3.2 Data registers in the processing unit of a System/360-370 computer

the register is utilized. There are 16 general-purpose registers in the processing unit, which are similar to the AC register of the PDP-11 computer, and these are numbered 0 to 15. These registers can be used as double-word registers, where the first register of a register pair is even-numbered.

3.3 ARITHMETIC INSTRUCTIONS

To make use of the flexibility of the memory organization, there is essentially a separate instruction repertoire for each datum length. The same is true for the PDP-11, but discussion of PDP-11 byte instructions is deferred until Chapter 6. We shall discuss in some detail the instructions that manipulate full-word data and then consider briefly the classes of instructions that manipulate data of other lengths.

Full-word data can be retrieved from memory and placed in one of the 16 general-purpose registers by using the load instruction (code L) as follows. The instruction L 3,X copies the full word, beginning at memory address X, into register 3. Of course, we assume that X is a memory ad-

dress that lies on a full-word boundary. The action of the L instruction is then

```
REG[3] ← MEMORY[X;X+1;X+2;X+3];
```

The four subscripts of MEMORY separated by semicolons indicate that four bytes of memory are transferred. The load instruction is comparable to the PDP-11 MOV. The leftmost eight bits of register 3 are replaced by the byte with memory address X, the next eight bits by byte X+1, and so forth. The following table gives the full-word instructions for arithmetic operations:

INSTRUCTION	CODE	EXAMPLE	ACTION
Load full word	L	L 3,X	REG[3] ← MEMORY [X;X+1;X+2;X+3];
Store full word	ST	ST 3,Y	MEMORY[Y;Y+1;Y+2;Y+3] ← REG[3];
Add full word	A	A 7,Z	REG[7] ← REG[7]+MEMORY [Z;Z+1;Z+2;Z+3];
Subtract full word	S	S 7,W	REG[7] ← REG[7]−MEMORY [W;W+1;W+2;W+3];
Multiply full word	M	M 6,U	REG[6,7] ← REG[7] ×MEMORY [U;U+1;U+2;U+3];
Divide full word	D	D 6,V	REG[7] ← REG[6;7] ÷ MEMORY [V;V+1;V+2;V+3]; REG[6] ← remainder of division;

The ST instruction is the inverse of the L instruction, and it is also similar to the MOV instruction for the PDP-11 computer. The A instruction adds two full-word operands and produces a full-word result; its PDP-11 counterpart is the ADD instruction. Note that the operands in a System/360-370 instruction are ordered not by source and destination as in the PDP-11, but rather by their location; namely, the operand that resides in a register is listed first and the operand that resides in memory is listed second.

The S instruction subtracts one full-word operand from another and places the difference in a processing-unit register. This corresponds to the SUB instruction in the PDP-11. As indicated in Chapter 2, the subtraction operations depend on the representation of negative numbers. System/360-370 uses two's-complement representation for the fixed-point full-word arithmetic operations, and the subtraction, multiplication, and division processes are clearly sensitive to this choice. Conse-

quently, the programmer cannot choose the representation of negative numbers flexibly, but must use two's-complement representation to obtain maximum efficiency.

The next two instructions, M and D, are not similar to instructions in the basic PDP-11, although comparable instructions may be added by purchase of special options. The M and D instructions require some discussion because of several characteristics not detailed in the table. In general, when two m-bit numbers are multiplied, the product can contain as many as $2m$ significant bits, but no more. For this reason the M instruction produces a double-word product from two full-word operands. We recall that the first register of a double-word register pair must be even-numbered, so that the result register—that is, the register addressed by the M instruction—must be even. This can be confusing, since the multiplier is not in the register addressed by the instruction but in the register with an index one greater.

As an example of the use of the M instruction, suppose that the full word beginning at MEMORY[X] contains $2^{18} + 1 = 0004\ 0001_{16}$ and suppose that REG[5] contains $2^{15} - 1 = 0003\ FFFF_{16}$. Then, after the instruction

```
M 4,X
```

The double-word register REG[4;5] will contain

$$2^{36} - 1 = 0000\ 000F\ FFFF\ FFFF_{16}$$

Naturally, the contents of REG[4] before the multiplication are lost, and they do not affect the result of the operation.

Division generally presents a problem in digital computers because a quotient may require an infinite number of digits in its representation even when both the divisor and the dividend involve a finite number of digits. Hence division is usually truncated to a fixed number of digits in the quotient. System/360-370 full-word division is integer division. By this we mean that both the dividend and the divisor are treated as integers, and an integer quotient is produced. Obviously the quotient cannot be exact except when the dividend is a multiple of the divisor. We indicate integer division in the description of the D instruction by using the symbol $\div$ instead of the symbol / as the division operator. Integer division is so defined that the remainder after division has the same sign as the dividend and is smaller in absolute value than the divisor. Thus $5 \div 3 = 1$ and $-5 \div 3 = -1$. In effect, positive quotients are truncated downward and negative quotients are truncated upward.

We noted that the product of two m-bit numbers will, in general, have $2m$ bits. By similar reasoning it is clear that the quotient of an

integer division involving a $2m$-bit dividend and an m-bit divisor will have as many as m bits. Hence the D instruction operates on a double-word dividend and a full-word divisor to produce a full-word result. For this reason the result register specified by the D instruction must be even. When the division operation is complete, the quotient is in the odd-numbered register and the remainder is in the even-numbered register of the same pair. Note that m bits may be insufficient to hold a quotient if the divisor is relatively small. For example, division of a $2m$-bit integer by the integer 1 produces a $2m$-bit quotient, which will not fit in an m-bit register. When this happens an overflow condition is recorded.

As an example of the D instruction, suppose that the full word beginning at MEMORY[X] contains the number $0000\ 0003_{16}$ and that the double-word register REG[8;9] contains the number

$0000\ 0000\ 0000\ 0005_{16}$

Then the instruction

```
D 8,X
```

produces the quotient $0000\ 0001_{16}$ in REG[9] and the remainder $0000\ 0002_{16}$ in REG[8].

If both operands of an arithmetic operation are available in the general-purpose registers, then it is possible to perform the arithmetic operation without fetching data from memory. The following full-word arithmetic instructions manipulate data stored only in the general-purpose registers:

INSTRUCTION	CODE	EXAMPLE	ACTION
Load (register)	LR	LR 3,4	REG[3] ← REG[4];
Load complement (register)	LCR	LCR 3,6	REG[3] ← −REG[6];
Add (register)	AR	AR 4,5	REG[4] ← REG[4]+REG[5];
Subtract (register)	SR	SR 2,7	REG[2] ← REG[2]−REG[7];
Multiply (register)	MR	MR 4,9	REG[4;5] ← REG[5] ×REG[9];
Divide (register)	DR	DR 6,3	REG[7] ← REG[6;7] ÷ REG[3]; REG[6] ← remainder;

Except for LCR, all these instructions are identical to the corresponding instructions that reference memory for one operand. The LCR instruction forms the two's complement of a number, so that it is used

to compute the negative of a number. LCR is comparable to the PDP-11 single-operand NEG instruction. The two registers referenced in the LCR instruction may be the same register.

There is a small set of instructions for halfword data that is similar to the full-word instruction set. These instructions are given in the following table:

INSTRUCTION	CODE	EXAMPLE	ACTION
Load halfword	LH	LH 3,W	REG[3] ← MEMORY[W;W+1];
Store halfword	STH	STH 5,X	MEMORY[X;X+1] ← REG[5];
Add halfword	AH	AH 6,Y	REG[6] ← REG[6]+MEMORY [Y;Y+1];
Subtract halfword	SH	SH 4,Z	REG[4] ← REG[4]−MEMORY [Z;Z+1];

The AH and SH instructions perform 32-bit two's-complement addition and subtraction in the registers even though the operand fetched from memory is only 16 bits in length. They do this by treating the leading bit of a 16-bit operand as a sign bit and generating the 32-bit representation that represents the same number for use in the arithmetic operation. This means that, if the sign bit is 0, the leading 16 bits of the 32-bit operand are cleared to 0 and, if the sign is 1, the leading 16 bits of the 32-bit term are set to 1. Thus the sign bit is extended through the leading 16 bits. Similarly, the LH instruction either clears to 0 or sets to 1 the 16 leading bits of the result register.

As mentioned earlier, floating-point operations are performed in the floating-point registers. There are instructions for manipulating both full-word and double-word floating-point numbers. For example, the following instructions illustrate four ways of loading a floating-point register (FPREG denotes a floating-point register):

INSTRUCTION	CODE	EXAMPLE	ACTION
Load (floating-point full word)	LE	LE 4,X	FPREG[4] ← MEMORY[X;...];
Load (floating-point double word)	LD	LD 2,Y	FPREG[2;3] ← MEMORY[Y;...];
Load (floating-point full word, register)	LER	LER 2,4	FPREG[2] ← FPREG[4];
Load (floating-point double word, register)	LDR	LDR 4,6	FPREG[4;5] ← FPREG[6;7];

The table illustrates that the suffixes E, D, ER, and DR are used to distinguish the four floating-point load instructions. The same suffixes are used, together with A, S, M, and D, to generate four versions of each of the floating-point arithmetic instructions. Similarly, STE and STD are the two floating-point store instructions.

We shall not consider floating-point operations and several other forms of arithmetic operation in detail here. Instead we shall limit our attention exclusively to the full-word instructions, since they illustrate the salient characteristics of the System/360-370 computers.

3.4 CONDITIONAL AND UNCONDITIONAL BRANCHING

Changing of program control flow in the System/360-370 is accomplished by essentially the same type of branching instructions as found in the PDP-11. Conditional branching is controlled by a two-bit register called the condition code. As in the PDP-11 the value of the condition code is determined by data-dependent conditions that exist when the execution of an instruction is completed. Such a condition as overflow, the sign of a result, or the relative magnitude of two operands can cause the condition code to assume a particular value. Unconditional branching is essentially the same as conditional branching, except that the condition code does not condition the branch.

The following instructions cause the condition code to be set to the values indicated:

INSTRUCTION	CODE	EXAMPLE	CONDITION CODE SETTING			
			0	1	2	3
Compare	C	C 3,X	REG[3] = MEMORY[X]	REG[3] < MEMORY[X]	REG[3] > MEMORY[X]	
Compare (register)	CR	CR 3,4	REG[3] = REG[4]	REG[3] < REG[4]	REG[3] > REG[4]	
Load and test (register)	LTR	LTR 3,4	REG[3] = 0	REG[3] < 0	REG[3] > 0	

The action of this example of LTR is to load REG[3] from REG[4] and then set the condition code. The LTR instruction is similar to the LR instruction in its execution; the only difference is that LTR sets the condition code and LR does not.

Many of the instructions that we have already discussed also set

the condition code. For example, the A instruction leaves the condition code set to reflect the outcome of an addition. The settings for this instruction are the following:

CONDITION CODE	MEANING
0	Sum is 0.
1	Sum is negative.
2	Sum is positive.
3	Overflow.

A listing of the condition code settings for all the instructions appears in the manual "Principles of Operation," form A22-6821 of the System/360 library. All our examples use only the C, CR, and LTR instructions to control conditional branches.

Conditional branches are executed by means of an instruction that contains a four-bit field called a *mask*. The four bits correspond to the four possible states of the condition code, as given in the following table:

CONDITION CODE	MASK BIT
0	$1000_2 = 8_{16}$
1	$0100_2 = 4_{16}$
2	$0010_2 = 2_{16}$
3	$0001_2 = 1_{16}$

When a conditional branch instruction is executed, the sequence of actions is as follows:

1. Find the value of the condition code.
2. Examine the corresponding bit of the mask.
3. If the mask bit is 1, then the branch is taken; otherwise, no branch occurs and the instruction immediately following the current instruction is the next to be executed.

The branch instructions that make use of the condition code are the following:

INSTRUCTION	CODE	EXAMPLE	ACTION
Branch on condition	BC	BC MASK,X	Take next instruction starting at MEMORY[X] if value of condition code specifies a mask bit with value 1.
Branch on condition (register)	BCR	BCR MASK,2	Take next instruction starting at MEMORY[REG[2]] if value of condition code specifies a mask bit with value 1.

The difference between BC and BCR is that the second operand of the BC instruction is a memory address, whereas the second operand of the BCR instruction is a register that contains a memory address.

Now suppose the instruction that immediately precedes a branch instruction is

```
CR 2,3
```

If the mask for the branch instruction is 8_{16}, then the branch instruction is taken only if registers 2 and 3 contain the same value. That is, the branch is essentially a branch if equal instruction. Therefore, instead of using the instruction

```
BC 8,X
```

it is more informative to write

```
BE X
```

where the BE means branch if equal. Of course, there is no BE instruction in the repertoire. Wherever the BE instruction appears, it should be replaced by BC with a mask equal to 8_{16}. Similarly, BC with a mask set to all 1's always causes a branch. Thus we use B for this case and call it branch unconditionally. The following table gives all branches that are of use to us. They are arranged in two groups, one for use after the C and CR instructions and the other for use after the LTR or arithmetic instructions. The table includes only the various forms of the BC instruction. There is a similar set that comprises variants of the BCR instruction. Following the usual convention, these have mnemonic codes that have the suffix R.

INSTRUCTION	CODE	MASK
Branch unconditionally	B	$F_{16} = 1111_2$
Branch on high	BH	$2_{16} = 0010_2$
Branch on low	BL	$4_{16} = 0100_2$
Branch on equal	BE	$8_{16} = 1000_2$
Branch on not high	BNH	$D_{16} = 1101_2$
Branch on not low	BNL	$B_{16} = 1011_2$
Branch on not equal	BNE	$7_{16} = 0111_2$
Branch on plus	BP	$2_{16} = 0010_2$
Branch on minus	BM	$4_{16} = 0100_2$
Branch on zero	BZ	$8_{16} = 1000_2$
Branch on not plus	BNP	$D_{16} = 1101_2$
Branch on not minus	BNM	$B_{16} = 1011_2$
Branch on not zero	BNZ	$7_{16} = 0111_2$
Branch on overflow	BO	$1_{16} = 0001_2$

There are two other classes of branch instructions that deserve mention here. One class, comprising the BCT and BCTR instructions, controls iterations of a set of instructions and corresponds to the INC instruction of the PDP-11 computer. The other class, comprising the BAL and BALR instructions, causes the address of the immediately following instruction to be saved before a branch is taken. The following table gives the special branch instructions. (Recall that the P register holds the address of the instruction being executed.)

INSTRUCTION	CODE	EXAMPLE	ACTION
Branch on count	BCT	BCT 5,X	REG[5] ← REG[5]−1; **if** REG[5] ≠ 0 **then** P ← X;
Branch on count (register)	BCTR	BCTR 5,6	REG[5] ← REG[5]−1; **if** REG[5] ≠ 0 **then** P ← REG[6];
Branch and link	BAL	BAL 14,X	REG[14] ← P+1; P ← X;
Branch and link (register)	BALR	BALR 14,R	REG[14] ← P+1; **if** R ≠ 0 **then** P ← REG[R];

The comments in the ACTION column are not totally accurate for reasons relating to the encoding of instructions. Where the phrase P+1 appears, it refers to the address of the instruction that follows. In reality the P register is increased by either 2 or 4, depending on the number of

bytes occupied by an instruction. Similarly, it is not completely accurate to say that just the P register is saved during the execution of the BAL and BALR instructions, since the condition code and other items also are saved. We shall discuss both these points later.

One last point of great importance is that the BALR instruction need not cause a branch. The explanation of the instruction indicates that the instruction

```
BALR 14,0
```

does not cause a branch to be taken, whereas

```
BALR 14,15
```

causes the next instruction to be taken from the memory cell at the address that is contained in REG[15]. In either case the address of the instruction that follows the BALR instruction is placed in REG[14].

3.5 INSTRUCTION ENCODING

All the instructions described thus far fall into one of two classes: those that specify a register and a memory cell or those that specify two registers. Since there are 16 general-purpose registers, only four bits are needed to specify a register. Clearly a four-bit memory address would be unduly restrictive. Consequently, instructions that contain memory addresses require more bits for encoding than equivalent instructions that have register references in place of memory addresses.

The encodings for the two classes of instructions are shown in Figure 3.3. Instructions that specify two registers are generally called RR instructions (register-to-register), and the instructions that specify a register and a memory cell are called RX instructions (register-to-indexed-storage). The encodings shown in Figure 3.3 indicate that RR instructions occupy two bytes of memory and the RX instructions occupy four bytes because of the extra room required for memory addresses.

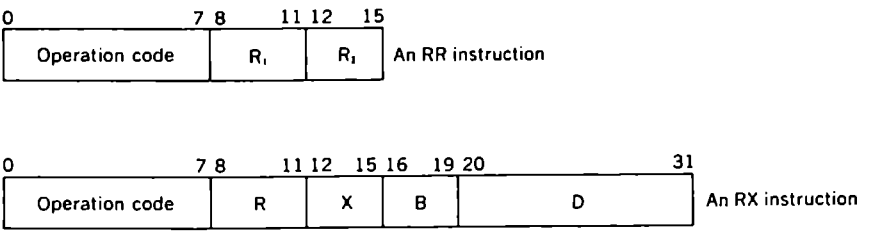


FIGURE 3.3 Instruction encoding for System/360-370 instructions

For all instructions the first byte is the *operation code*, which uniquely identifies the instruction. In RR instructions the next byte contains the identifiers of two registers, each four bits in length. The RX instructions use the second byte slightly differently than the RR instructions use it. The first four bits contain a register identifier, and the second four bits, labeled X in Figure 3.3, are used in conjunction with the remaining portion of the instruction to specify a memory address. The next two bytes contain a four-bit segment labeled B and a 12-bit segment labeled D. The contents of the B, D, and X fields specify a memory address according to an algorithm that will be described in detail in Chapter 4.

3.6 THE PROGRAM STATUS WORD

System/360-370 computers have a 64-bit register called the *program status word* that holds information relating to the state of the computer. Obviously the address of the instruction currently being executed should be included in this register, and, indeed, Figure 3.4 shows that the P register is the rightmost three-byte portion of the program status word. Note also that the condition code resides in bits 34 and 35 of the program status word. Other portions of the program status word contain information that is not relevant to our discussion here.

The idea behind the program status word is to concentrate all the information pertaining to the state of a program into one register. Then one program can be suspended and another can be initiated by changing the contents of the program status word, provided that both programs can be resident in memory simultaneously. There are instructions both for storing and for reloading the program status word that facilitate the process of changing from one program to another.

Any instruction that can modify the entire program status word can be executed only by a program called the *operating system* or *executive program*, since this is the program that controls the allocation of the central processing unit among several programs. The mode of operation in which a computer can be shared by several programs that reside in memory simultaneously is generally known as *multiprogramming*.

Ordinary programs can access the rightmost four bytes of the pro-

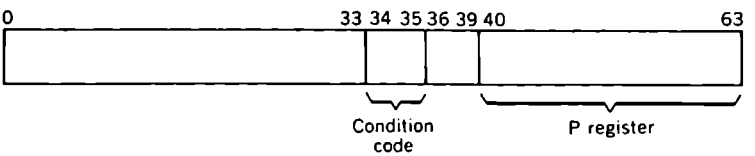


FIGURE 3.4 The program status word for System/360-370. The unlabeled fields contain miscellaneous control information

gram status word by means of the BAL and BALR instructions. As indicated earlier, these instructions cause the P register to be copied into a designated register. Actually, the entire four-byte group that contains the P register and the condition code is copied into the designated register. Our concern, however, is only in the ability to access the P register.

REFERENCES

The description of a large-scale computer such as the System/360-370 is inherently encumbered by myriad facts, any one of which may be essential for treating a pathological programming situation. The most complete source of all relevant facts is the IBM Systems Reference Library, but its completeness interferes somewhat with efficient retrieval of pertinent information. Within this library, "Principles of Operation," form A22-6821, contains a complete description of the System/360 computer organization and of the instruction repertoire. It is a good reference document and serves well as a supplement to this text.

FOUR

ADDRESSING MODES AND LOGICAL OPERATIONS

Earlier chapters covered basic arithmetic instructions that are common to many computers. However, arithmetic computations represent a very small subset of the many types of computations for which computers are well suited. The Turing machine view suggests that computers are primarily symbol manipulators. Arithmetic computations are merely computations in which the symbols have meaning as numbers. Computers are also extremely valuable for such basically nonnumeric operations as information storage and retrieval, graphic arts, and simulation.

The logical operations and the addressing mechanisms described in this chapter significantly enhance the symbol-processing power of computers. Later chapters illustrate algorithms that make use of these processing capabilities.

4.1 ADDRESSING MODES

Computers normally have facilities that permit the programmer to specify addressing modes for the operands of an instruction when the instruction is executed. There are several applications for addressing modes that will come to light in later examples. One very important application is worth mentioning here. Addressing modes permit a single sequence of instructions to be used iteratively, acting upon different data in each iteration. Several different addressing modes have been implemented in computers, and no single set of modes is common to all computers. Addressing modes in the PDP-11 computer are quite different from those in System/360-370 computers, for example.

With few exceptions, addressing modes are implemented in the following way. The encoded form of an instruction includes an operand address plus other data that indicate how the address is to be modified when the instruction is executed. When an instruction is executed, the following actions occur:

1. The instruction is fetched from memory.
2. The operand address is extracted together with the addressing-mode data.
3. The computer executes an algorithm that uses the addressing-mode data and the operand address to produce a new operand address.
4. The new operand address is the address used for the execution of the instruction.

The address produced by the addressing mode algorithm is called the *effective address* of the instruction. Address modification never alters the stored instruction in memory; it only modifies the interpretation of the instruction at the time of execution.

The sections that follow treat several different addressing modes that are available in the PDP-11 and System/360-370 computers.

4.1.1 Relative Addressing

One of the simplest ways to perform addressing is to provide the address, or part of the address, in the instruction itself. The addressing mode specified by the bit pattern 110 111 in the source field of an instruction is interpreted by the PDP-11 to mean that the effective address of the source operand is calculated by adding the contents of the P register to

the word following the instruction. We call this word an *address word.* The effect of the instruction MOV X,AC is:

```
MAR ← P;
P ← P+2;
comment fetch the address word;
MDR ← MEMORY[...;MAR];
comment compute the source address;
MAR ← MDR+P;
comment fetch the source operand to AC;
AC ← MEMORY[...;MAR];
```

The address word of the MOV instruction thus contains a symbol that represents the number of memory cells away from the current value of the P register the operand is to be found. Here the value stored in the address word is X−(P+2), where X is the memory address of the operand. The 2 appears because the P register is incremented after the address word is fetched and before the address calculation. This form of addressing is said to be *relative to the program counter,* and the value X−(P+2) stored in the address word is called the *relative address* of X. In the PDP-11 any addressing mode can be used for the destination operand as well as for the source operand. Thus it is possible that the addressing mode of both the source and destination operand is relative to the program counter. In this case the instruction has two address words, one for the source and one for the destination, and the effective address calculations are identical. Note the instruction in this instance occupies three words. For example, the effect of the instruction MOV Y,X is:

```
comment fetch the second word to compute the source address;
MAR ← P;
P ← P+2;
MDR ← MEMORY[...;MAR];
MAR ← MDR+P;
comment the source address is now in MAR. Fetch the source operand;
MDR ← MEMORY[...;MAR];
SOURCE ← MDR;
comment SOURCE now contains the source operand. Now compute the
destination address;
MAR ← P;
P ← P+2;
MDR ← MEMORY[...;MAR];
```

```
MAR ← MDR+P;
comment MAR now contains the address of X. Execute the MOV;
MEMORY[. . . ;MAR] ← SOURCE;
```

In any event, the source or destination address is calculated as A+(P+2), where A is the value in the address word and P is the address of the address word.

Relative addressing is a very powerful addressing mode. The relative address words do not have to be changed if a program is moved from one place in memory to some other, since the relative position of instructions and data is unchanged. For example, consider what happens if the program containing the MOV X,AC is moved in memory. The addresses of the memory cells in which the MOV instruction and the operand X are actually stored change. But the relative address in the address word of the MOV instruction does not change. Instructions using the relative addressing mode are said to be *relocatable*, since they need not be modified if the program of which they are part is to be executed in physically different places in memory.

Branch instructions also use relative addressing. The offset field contains the number of *words* the branch destination is displaced from the current value of the P register. Since the P register contains byte addresses and not word addresses, the offset must be multiplied by 2 before addition to the P register. The reason for multiplication by 2 is that instructions must begin at even byte addresses. The multiplication doubles the range of destination addresses that can be designated by the offset and guarantees that an odd destination address cannot be generated through program error.

4.1.2 Register Addressing

The bit pattern 000 000 in the source field indicates in the PDP-11 that the operand is in the AC. To load the AC from MEMORY[. . . ;X] we use MOV X,AC encoded as in Figure 4.1*a*, whereas to store the AC in MEMORY [. . . ;X] we use MOV AC,X, which is encoded as in Figure 4.1*b*.

4.1.3 Deferred Addressing

Both the register and relative addressing modes in the PDP-11 are used to calculate an effective address. It is frequently advantageous to be able to fetch an operand with a single instruction when the operand address is known to be stored in MEMORY[. . . ;X]. To accomplish this we use an addressing mode known as *deferred* or *indirect* addressing. We indicate deferred addressing by the symbol @. The following list of instructions shows the effect of deferred addressing:

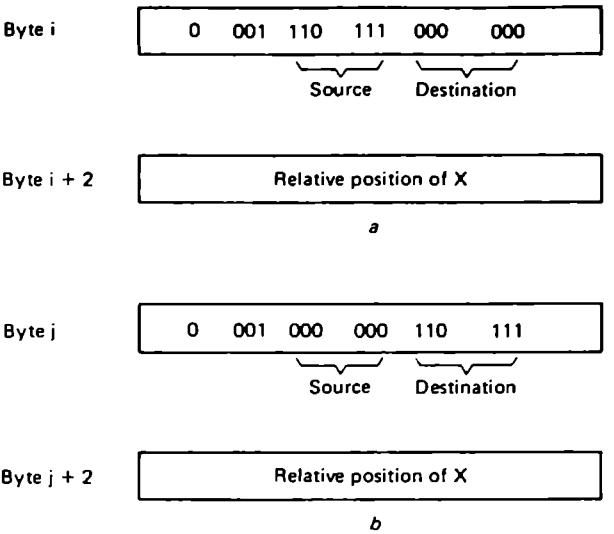


FIGURE 4.1 Example instruction encoding for (*a*) MOV X,AC and (*b*) MOV AC,X

INSTRUCTION	ACTION
MOV @X,AC	AC ← MEMORY[MEMORY[. . . ;X]]
MOV X,@AC	MEMORY[. . . ;AC] ← MEMORY[. . . ;X]
ADD @X,AC	AC ← MEMORY[MEMORY[. . . ;X]]+AC
MOV AC,@X	MEMORY[MEMORY[. . . ;X]] ← AC

Note that @X forces the contents of MEMORY[. . . ;X] to be treated as the address of an operand rather than as an operand itself. Similarly, @AC forces the contents of register AC to be treated as the address of the operand. The @ signifies this notion and denotes one more memory reference in the effective address calculation than without it. This last reference is the *deferred* or *indirect* reference.

Several different forms of addressing are shown for the INC instruction below.

EXAMPLE	ACTION	DESCRIPTION
INC AC	AC ← AC+1	Level 0, register addressing
INC X	MEMORY[. . . ;X] ← MEMORY[. . . ;X]+1	Level 1, direct addressing
INC @AC	MEMORY[. . . ;AC] ← MEMORY[. . . ;AC]+1	Level 2, deferred addressing
INC @X	MEMORY[MEMORY[. . . ;X]] ← MEMORY[MEMORY[. . . ;X]]+1	Level 2, deferred addressing

Now, as an example of deferred addressing, let us simulate a Turing machine with the PDP-11 computer. The tape of the Turing machine will be simulated by computer memory such that successive *words* in memory represent successive squares on the Turing machine tape. In this simulation the tape alphabet is {0,1}.

Each Turing machine instruction is a five-tuple consisting of a state q_i, current symbol s_j, next state q_{ij}, written symbol s_{ij}, and head direction d_{ij}. For each state q_i in the Turing machine to be simulated we shall use addresses of the form QI, QIONE, QIZERO. QI is the address of the first instruction of a sequence of instructions that simulate the machine in state q_i. QIONE and QIZERO are the addresses of instructions that simulate the behavior of the machine in state q_i with a 1 input and a 0 input, respectively. The current position of the head is marked by a memory address placed at the symbolic address HEAD. The following instructions direct the simulation of a machine in state q_i to either QIONE or QIZERO, depending on the symbol that is currently being scanned by the head. Note that both source and destination operands of the CMP instruction are memory cells.

```
ADDRESS   CODE   OPERAND     COMMENTS
QI:       CMP    @HEAD,ONE   Compare current symbol to 1.
          BEQ    QIONE       If equal, jump to QIONE; other-
QIZERO:   ...                wise continue to QIZERO.
QIONE:
```

At addresses QIONE and QIZERO the instructions simulate the action part of a five-tuple. There are three different actions to take. First we rewrite a symbol on the tape with

```
MOV ONE,@HEAD
```

or

```
MOV ZERO,@HEAD
```

where ZERO and ONE are the symbolic addresses of words in memory that contain the numbers 0 and 1, respectively. Next we move the tape head left or right with the following instructions, recalling that symbols are stored one per memory word and the memory is byte addressable:

```
ADD TWO,HEAD    Move head right. HEAD ← HEAD+2.
SUB TWO,HEAD    Move head left. HEAD ← HEAD—2.
```

Finally, we enter a new state, q_j:

```
BR QJ
```

Example 4.1 is a simple simulation of the Turing machine program in Example 1.2.

EXAMPLE 4.1

ADDRESS	CODE	OPERAND	COMMENTS
Q1:	CMP	@HEAD,ONE	Compare next symbol to 1.
	BEQ	Q1ONE	If it is a 1, branch.
Q1ZERO:	MOV	ONE,@HEAD	If it is a 0, write a 1 and halt.
	BR	STOP	
Q1ONE:	MOV	ONE,@HEAD	Rewrite the 1.
	ADD	TWO,HEAD	Move head to right.
	BR	Q1	Iterate.
STOP:	HALT		HALT stops the computer.

4.1.4 Additional Addressing Modes

Thus far we have described addressing modes known as register, relative, and deferred. Several other modes are available in the PDP-11, and these are representative of modes available in most computers. In this section we discuss the full set of PDP-11 addressing modes, but we leave detailed examples of their use until later. In the following section we treat the IBM/360-370 addressing modes, which are similar to the PDP-11 modes but differ in specific details.

In order to give the full details of PDP-11 addressing, we describe some facets of the PDP-11 computer organization that have heretofore been omitted to simplify discussion. We have treated the PDP-11 as if it had a single accumulator called AC, but in reality the PDP-11 has eight accumulators which we denote as REG[0], REG[1], . . ., REG[7]. REG[0] is AC in this convention. An accumulator is a processor register whose major function is to hold operands. An accumulator is identical in its physical implementation to the P register, MDR, and MAR; in fact, in the PDP-11 computer the P register is REG[7]. We make use of this fact in some of the addressing modes. If a source operand is in an accumulator, we do not

need to fetch it from memory, and thus we save a memory cycle. Consequently, it is advantageous to keep frequently used quantities in accumulators. With more accumulators in a processor, there is more opportunity to take advantage of this saving, which suggests why the PDP-11 has multiple accumulators.

Apart from the modes described earlier there are three other major addressing modes for the PDP-11, namely, index, autoincrement, and autodecrement. As in all PDP-11 addressing modes, each of these modes utilizes one of the processor registers in the effective-address computation. Interesting variations are obtainable when REG[7], the P register, is used in an address calculation instead of one of the other registers.

In the index addressing mode the operand address is calculated as:

effective address ← REG[INDEX]+MEMORY[. . . ;P].

In this case, the register, REG[INDEX], is said to be an *index register*. Here we recall that the P register designates an address word of the instruction. The contents of the address word are added to REG[INDEX] to form the operand address, and the P register is incremented by 2. By convention, we specify index mode with parentheses as in the example instruction

INC X(INDEX)

which denotes REG[INDEX] as the index register and X as the contents of the address word of the instruction.

To see how this particular instruction is useful, assume X is the integer 100 and observe how the effective address of the instruction changes as REG[INDEX] passes through the values 0, 2, 4, . . ., 18. If the INC X(INDEX) is executed repeatedly for each of these values in REG [INDEX], the memory words at address 100, 102, . . ., 118 will each be incremented. Thus the INC X(INDEX) instruction can operate on many different data depending on the value stored in REG[INDEX]; yet the instruction itself is not modified in memory to obtain this effect. The address calculation during instruction execution determines the operand for the instruction, and the operand depends on the contents of REG[INDEX]. We can place INC X(INDEX) in a context in which REG[INDEX] passes through a predetermined sequence of values so that the INC X(INDEX) operates on a predetermined sequence of words in memory. We can update the words in the order 100, 102, . . ., 118, in the order 118, 116, . . ., 102, 100, or in fact in any order depending on how the contents of REG[INDEX] change during computation. Example 4.2 uses the index addressing mode to add 10 words in memory.

EXAMPLE 4.2

We wish to add the 10 words in memory, beginning at memory address 100_{10}. Assume that TEN is the symbolic address of the memory cell that contains the value 10.

	LABEL	CODE	OPERAND	COMMENTS
1.		MOV	TEN,COUNT	Initialize counter to 10.
2.		CLR	SUM	Initialize sum to 0.
3.		CLR	AC	AC ← 0; AC now contains index of first addend.
4.	LOOP:	ADD	100(AC),SUM	During each iteration a new effective address is produced here.
5.		ADD	TWO,AC	Increase AC by 2 to find next full word.
6.		DEC	COUNT	Decrease counter by 1. If it is not 0, branch to LOOP is taken.
7.		BGT	LOOP	
8.	NEXT:			When this point is reached, the 10 numbers have been added.

Obviously, the ascending sequence 100, 102, . . ., 118 and the descending sequence 118, 116, . . ., 102, 100 are address sequences that are likely to be the most useful. These can be generated in an especially simple way by making use of addressing modes called *autoincrement* and *autodecrement*.

The autoincrement addressing mode takes the following actions:

```
effective address ← REG[Rn];
REG[Rn] ← REG[Rn]+2;
```

where Rn denotes processor register REG[n] for $0 \leq n \leq 7$. Note that the autoincrement mode is the same as the register deferred addressing mode except that the register is automatically increased after the address calculation. We denote this mode symbolically as

```
INC (Rn)+
```

where the notation (Rn) is intended to signify the contents of Rn and the + indicates the updating of the register after the address calculation. Example 4.2 can be rewritten to use the autoincrement addressing mode as depicted in Example 4.3.

EXAMPLE 4.3

Assume that TEN and HUNDRED are the symbolic addresses of memory cells that contain the values indicated.

	LABEL	CODE	OPERAND	COMMENTS
1.		MOV	TEN,COUNT	Initialize counter to 10.
2.		CLR	SUM	Initialize sum to 0.
3.		MOV	HUNDRED,AC	AC ← 100.
4.	LOOP:	ADD	(AC)+,SUM	During each iteration a new effective address is produced in AC.
5.		DEC	COUNT	Decrease counter by 1. If it is not 0, branch to LOOP.
6.		BGT	LOOP	
7.	NEXT:	...		When this point is reached, the 10 numbers have been added.

Autodecrement mode is quite similar to autoincrement. In autodecrement mode the following actions occur.

```
REG[Rn] ← REG[Rn]−2;
effective address ← REG[Rn];
```

By convention we denote autodecrement as in the example instruction INC −(R2). Note that the register is modified *after* an autoincrement operation and *before* an autodecrement operation. The notation used here symbolizes this by placing the minus sign before the address expression. With the autodecrement mode, Example 4.3 can be modified easily to process the memory addresses 118, 116, . . ., 102, 100 in that order. This particular example is insensitive to the order in which the addresses appear, but it is not difficult to construct realistic examples of programs in which ascending order, or descending order, must be followed, and for such programs it is extremely convenient to have autoincrement and

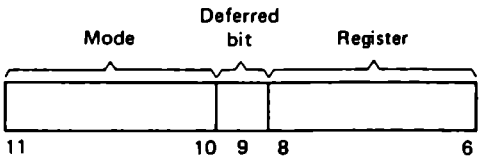


FIGURE 4.2 The subfields of a source field for the PDP-11

autodecrement addressing modes. Such examples appear in Chapters 6 and 7.

Figure 4.2 illustrates a source field for an instruction to show the encoding of the addressing modes. The leftmost bits, bits 11 and 10, specify one of the four modes: register, autoincrement, autodecrement, or index. The next bit, bit 9, specifies whether the mode is direct or deferred. Each of the four modes specified by the first two bits can be either direct or deferred, so that there are eight addressing modes in total. The last three bits, bits 8 to 6, specify which register is used in the effective address calculation. These modes and their encodings are shown in Table 4.1. Encoding of the destination field follows the same rules as for the source field. In each case, the deferred variant of an address mode performs the same address calculation as the direct variant, except that the deferred mode has a deferred address-reference at the end of the address calculation. Register references in the table are denoted by the symbol % followed by an integer n, where $0 \leq n \leq 7$. This follows the

TABLE 4.1

ADDRESSING MODE	SYMBOLIC NOTATION	ENCODING	EXAMPLE
Register	%n	000 n	%3
Autoincrement	(%n)+	010 n	(%2)+
Autodecrement	−(%n)	100 n	−(%4)
Index	X(%n)	110 n	BASE(%3)
Register, deferred	@%n	001 n	@%3
Autoincrement, deferred	@(%n)+	011 n	@(%0)+
Autodecrement, deferred	@−(%n)	101 n	@−(%1)
Index deferred	@X(%n)	111 n	@MAX(%4)
Immediate	#X	010 111	#23 or #MAX
Absolute	@#X	011 111	@#100 or @#PLACE
Relative	X	110 111	PLACE
Relative deferred	@X	111 111	@POINT

convention established for the PDP-11, and we use it here to reinforce the difference between register and memory operands. An X in the second column denotes an operand of the instruction that is encoded in an address word.

We mentioned that very interesting modes can be constructed from the basic modes if we designate REG[7], the P register, as the register in the address calculation. In fact, we recall that the encoding of the relative addressing mode is 67_8, which specifies index mode with the P register as index register. Indeed, this is precisely how we defined the relative addressing mode, except for the offset of 2 to account for modification of the P register. The other modes that use REG[7] are deferred relative, which has been discussed earlier, and two new modes called *immediate* and *absolute*. Immediate addressing uses the address word of the instruction as an operand. It is called immediate because the operand is part of the instruction rather than remote from the instruction and is thus "immediately" available when the instruction is fetched for execution. In many computers immediate operands can be obtained directly from the instruction without any additional memory references, but the PDP-11 requires a memory reference to obtain the immediate operand from the address word of the instruction. The immediate addressing mode is actually the autoincrement mode with REG[7] as the specified register. The absolute addressing mode forces the address word of the instruction to contain the address of the operand rather than a relative address or some other address from which the operand address can be calculated. The absolute addressing mode is the autoincrement deferred mode using REG[7].

Our description of the PDP-11 processor given in Chapter 2 must be modified as shown below to include the extra addressing modes. Note that the source *data* are always available in the SOURCE register at the end of the source state and that the destination *address* is in the DEST register after the DESTINATION state. This description, like the shortened version of it in Chapter 2, indicates that the eight machine registers have memory addresses. In fact REG[n] is assigned address $777700_8 + n$ for $0 \leq n \leq 7$. Even though they have memory addresses, no memory operation occurs when a register is a source or destination, and thus instructions with register operands require less time to execute than instructions with memory operands.

```
FETCH state:    MAR ← P;
                MDR ← MEMORY[...;MAR];
                P ← P+2;
                I ← MDR;
                  .
                  .
                  .
```

SOURCE state:

```
if I11–10 = 00 then
    begin comment register mode;
        SOURCE ← REG[I8–6];
    end
else
if I11–10 = 01 then
    begin comment autoincrement mode;
        MAR ← REG[I8–6];
        SOURCE ← MEMORY[. . . ;MAR];
        REG[I8–6] ← REG[I8–6]+2;
    end
else
if I11–10 = 10 then
    begin comment autodecrement mode;
        REG[I8–6] ← REG[I8–6]–2;
        MAR ← REG[I8–6];
        SOURCE ← MEMORY[. . . ;MAR];
    end
else
if I11–10 = 11 then
    begin comment index mode;
        MAR ← P;
        P ← P+2;
        MDR ← MEMORY[. . . ;MAR];
        MAR ← MDR+REG[I8–6];
        SOURCE ← MEMORY[. . . ;MAR];
    end;
if I9 = 1 then
    begin comment deferred addressing;
        MAR ← SOURCE;
        SOURCE ← MEMORY[. . . ;MAR];
    end;
change state to DESTINATION;
```

DESTINATION state:

```
if I5–4 = 00 then
    begin comment register mode. The memory
    address of REG[n] is 777700₈+n;
        DEST ← 777700+I2–0;
    end
else
if I5–4 = 01 then
    begin comment autoincrement mode;
        DEST ← REG[I2–0];
        REG[I2–0] ← REG[I2–0]+2;
    end
else
if I5–4 = 10 then
    begin comment autodecrement mode;
        REG[I2–0] ← REG[I2–0]–2;
        DEST ← REG[I2–0];
else
if I5–4 = 11 then
    begin comment index mode;
```

```
                        MAR ← P;
                        P ← P+2;
                        MDR ← MEMORY[. . . ;MAR];
                        DEST ← MDR+REG[I2–0];
                    end;
                if I3 = 1 then
                    begin comment deferred addressing;
                        MAR ← DEST;
                        DEST ← MEMORY[. . . ;MAR];
                    end;
                change state to EXECUTE;

EXECUTE state:  if I15–12 = 0110 then
                    begin ADD instruction execution;
                        .
                        .
                        .
                        .
```

4.1.5 Index Registers and Base Registers

The System/360-370 computers use their general-purpose registers in a slightly different manner than the PDP-11. In the PDP-11 the general registers hold the operand, an operand address, or an increment that is to be added to an address word to obtain the operand address.

Instructions in the System/360-370 are encoded to contain a portion of the operand address called the *displacement.* (The displacement is in some ways comparable to the offset in the PDP-11 branch instructions or to the address word of an instruction specifying index addressing.) Address modification is accomplished by specifying a general-purpose register whose contents are to be added to the displacement to produce an effective address. Thus the effective address of an instruction can be modified by changing the contents of a general register.

The address-modification facilities pertain to the RX instructions, and not to the RR instructions, since the latter do not specify memory locations. (System/360-370 instruction formats are discussed in Section 3.5.) Consider the L instruction shown in Figure 4.3. We recall that the L instruction loads a register from a full word in memory. The first byte of the L instruction contains the operation code, which is 58_{16} for this case, and the R field contains the number of the result register. The fields labeled X, B, and D—for index, base, and displacement—enter into the

Operation code	R	X	B	D
0101 1000	0010	0011	0101	0000 0001 0000

FIGURE 4.3 An example of the L instruction

effective address calculations as follows. The register labeled S is used to compute the effective address.

```
S ← D;
If X ≠ 0 then S ← S+REG[X];
if B ≠ 0 then S ← S+REG[B];
comment S now contains the effective address;
```

When we write instructions in which we indicate the B, D, and X portions of an address explicitly, the convention normally followed is illustrated as follows for the L instruction:

```
L R,D(X,B)
```

EXAMPLE 4.4

Assume that REG[3] contains 1000_{10} and REG[5] contains 3000_{10}. Then the instructions below are executed as indicated:

INSTRUCTION	ACTION
L 2,16(0,5)	REG[2] ← MEMORY[3016; . . .];
L 2,16(3,5)	REG[2] ← MEMORY[4016; . . .];
L 2,16(5,0)	REG[2] ← MEMORY[3016; . . .];
L 2,0(0,3)	REG[2] ← MEMORY[1000; . . .];
L 2,100(0,0)	REG[2] ← MEMORY[100; . . .];

Note that there is no intrinsic difference in processing the X field and the B field of an instruction; but although the fields are treated identically, they are usually put to two different uses. When address modification is used so that an instruction can operate on different data each time it is executed, the address modification is said to be an *indexing operation* and the register used for address modification is said to be an *index register*. Address modification can also be used to relocate a program in memory, so that the program can be executed from more than one address. Address modification used in this manner is said to be a *program relocation*, and the register involved is called a *base register*. System/360-370 instructions permit indexing and relocation simultaneously. By convention, the X field usually specifies an index register

and the B field usually specifies a base register. We give examples of both uses later in this section.

The index, base, and displacement fields in the System/360-370 instructions can be used to achieve the same effect as some of the PDP-11 addressing modes. For example, the use of an index register in the System/360-370 is very similar to index addressing in the PDP-11. Example 4.5 depicts Example 4.2 as rewritten for the System/360-370 to use an index register.

EXAMPLE 4.5

We wish to add the 10 words in memory, beginning at memory address 100_{10}. Assume that TEN, FOUR, ZERO, and HUNDRED are the symbolic addresses of memory cells that contain the values indicated. For the System/360-370 the program is:

ADDRESS	CODE	OPERAND	COMMENTS
	L	COUNT,TEN	REG[COUNT] ← 10;
	L	SUM,ZERO	REG[SUM] ← 0;
	L	INDEX,HUNDRED	REG[INDEX] ← 100;
LOOP	A	SUM,0(INDEX,0)	
	A	INDEX,FOUR	Increase INDEX by 4 to move to next full word.
	BCT	COUNT,LOOP	COUNT is decreased by 1. If it is not 0, branch to LOOP is taken.
NEXT:			When this point is reached, the 10 numbers have been added.

Examples 4.2 and 4.5 show how address modification lets us use just one addition instruction to add 10 numbers. However, index and base registers are more flexible than is indicated by Example 4.5 because there are uses for which we need not set the B and D fields to 0.

Base registers have entirely different applications from index registers. When a general-purpose register is used as a base register, its contents remain constant for all instructions that use it as a base register. This is in contrast to index registers, where the power of indexing is achieved through the ability to change the contents of the register.

Earlier examples show how base registers produce effective addresses. A typical application of a base register is illustrated in Example 4.6.

EXAMPLE 4.6

Suppose that Z is the address of the first byte of a full-word operand and that Z is 100 bytes beyond the first byte of the instruction labeled LOOP. Then the following instructions can be executed from any address in memory for which LOOP begins on a full-word boundary. (Compare these instructions with their equivalents in Example 4.5.)

ADDRESS	CODE	OPERAND	COMMENTS
	BALR	15,0	Place address of LOOP in REG[15], no branch taken.
LOOP	L	2,100(0,15)	Loads MEMORY[Z; . . .] into REG[2].
	. . .		
	BCT	6,0(0,15)	Branch on count. If REG[6] $\neq$ 0, then branch back to LOOP, the instruction addressed by REG[15].

The index and base register operations are shown separately in Examples 4.5 and 4.6 for clarity. The usual case, of course, is to combine the operations, so that the L instruction would become

```
L 2,100(INDEX,15)
```

Note in Example 4.6 that the BALR instruction establishes REG[15] to be a base register, so that both LOOP and Z are effective addresses produced by base-register computations. The effective address will be the correct effective address no matter where the program is placed in memory, provided that LOOP begins on a full-word boundary.

As we have noted, in the System/360-370 computer there are 12 bits per instruction that are interpreted as a memory address, which permits any one of 4096 different memory locations to be specified. What happens when a program and its data occupy more than 4096 words of memory? The use of base registers allows an arbitrary 32-bit number to be added to the displacement.

We might consider why the designers of the System/360-370 did not use a longer address field in the instructions themselves—say, 20 bits, which would increase the number of addressable words to 1,048,576. Clearly, if the address field of an instruction contained 20 bits, the instruction would then contain 40 bits or more, and the size of the computer word would have to be increased to accommodate the longer instruction. The net effect, of course, would be to increase the cost of the memory. Hence designers are tempted to seek alternative techniques for extending the addressable size of memory that are considerably less expensive to implement than large address fields in instructions. The use of various addressing modes as in the System/360-370 and PDP-11 computers increases the flexibility of addressing at the cost of increased execution time, yet the size of an address field may be kept quite small.

4.1.6 Paging

Another mechanism that can be used to increase the amount of addressable memory is called *paging*. It is similar to the use of base registers for this purpose, but there are important differences. A simple form of paging is illustrated in Example 4.7 and Figure 4.4. The idea is to form an effective address, in this case a 16-bit address, by using normal address computations such as might be found in PDP-11 or System/360-370 com-

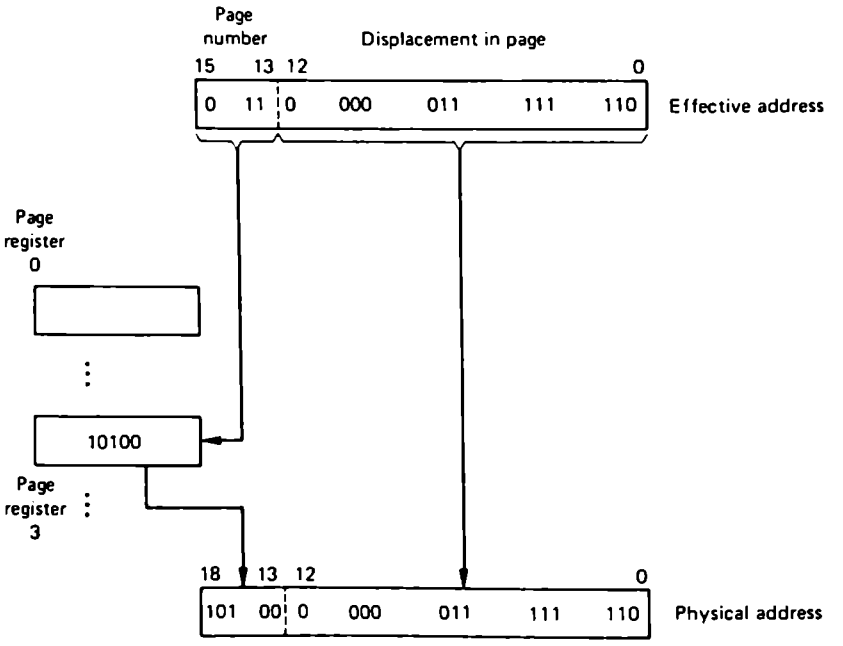


FIGURE 4.4 A paging computation with multiple page registers

puters. After the effective address is computed, the address is broken into two fields called *page number* and *displacement in page.* The page number selects one register from a set of registers called *page registers,* which are used to generate memory addresses from shorter effective addresses. The contents of the selected page register replace the page number in the effective address, thus creating a physical memory address quite different from the effective address produced by the program.

EXAMPLE 4.7

An instruction produces the 16-bit effective address 060376_8. The most significant three bits indicate page register 3. Page register 3 contains the 5-bit value 24_8. The physical memory address is computed to be the 18-bit quantity 500376_8. Figure 4.4 depicts this paging example.

In Figure 4.4 the page number contains three bits and is replaced by five bits stored in a page register. Thus paging has increased the size of a memory address by two bits, or equivalently, the size of addressable memory is increased by a factor of 4. The crucial difference between paging in this example and the action of base registers in the preceding section is that the contents of a page register are *concatenated* to a displacement whereas the contents of a base register are *added* to a displacement.

Although the differences between page registers and base registers appear to be slight, they have rather serious ramifications. The major issue is one of program relocation. The action of address modification performed by base and page registers not only increases the size of an address but displaces that address in memory. We recall that when base registers are used, a program can be placed anywhere in memory and will work correctly provided the base registers are initialized to the proper base addresses. (In System/360-370 computers, the program should be moved to a double-word boundary so that boundaries of halfwords, full words, and double words are maintained.) When paging is used, programs cannot be relocated to arbitrary places in memory. In Figure 4.4 for example, the 16-bit effective address can be changed into only 32 different physical memory addresses by a paging operation, because a page register can contain only numbers restricted to the interval $0 \leq x \leq 31$. In this case, if y is the displacement within a page and x is the contents of the page register, then the physical memory address is $x \cdot 2^{13} + y$. In Figure 4.4, x has the value $10100_2 = 20_{10}$.

Consider what happens as the displacement y ranges over all of the 2^{13} possible values. Then the physical memory addresses range over the 2^{13} contiguous addresses in the interval $20 \cdot 2^{13} \leq n \leq 21 \cdot 2^{13} - 1$. If we place a different 5-bit number in page register 3, then a different set of 2^{13} contiguous addresses is accessible. We call such a set of contiguous addresses a *memory page*. In Figure 4.4 we see that all effective addresses that lie in page 3 are redirected to a single physical memory page, in this case, physical memory page 20_{10}. Moreover, an effective address with a displacement y within page 3 of the program is directed to the word with displacement y within physical memory page 20.

To summarize the characteristics of paging, physical memory is broken into pages of size 2^{13} and the effective addresses of a program are likewise broken into pages of the same size. A program page will execute correctly provided the page registers contain the correct physical page numbers for all page numbers produced from effective addresses during execution. The partitioning of memory into pages is a direct consequence of the concatenation operation. The page size of 2^{13} in this example stems from the fact that the displacement field contains 13 bits.

Some models of the PDP-11 have an addressing mechanism similar to the paging mechanism described here but somewhat more flexible. The address computation for these models is shown in Figure 4.5 and Example 4.8. We see that 16-bit addresses are transformed to longer

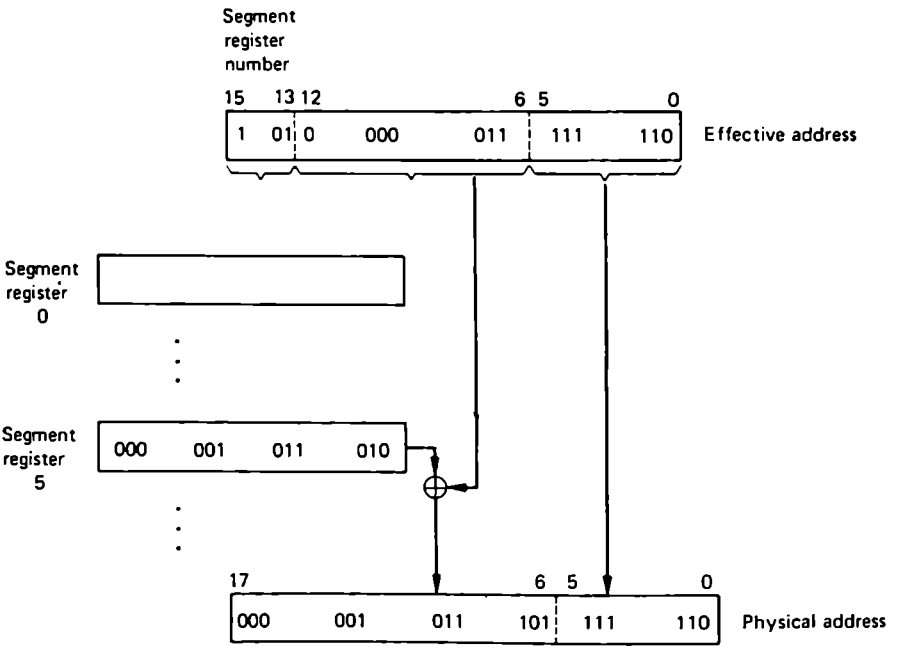


FIGURE 4.5 The PDP-11 physical address computation

18-bit addresses by this mechanism, but it is neither a paging nor a base register transformation as we have described them. The leading three bits of an effective address select one register from a set of registers whose contents are then added to the seven next most significant bits of the effective address. The least significant six bits are unchanged. Here we adopt the PDP-11 terminology and refer to the registers as *segment* registers and to the general technique as *segmentation*, although other meanings have been ascribed to segmentation.

EXAMPLE 4.8

The 16-bit effective address is 120376_8. The most significant three bits indicate segment register 5, which contains the 12-bit value 0132_8. The physical memory address is computed to be:

$(0132_8 + 003_8) \square 76_8 = 013576_8$

where $\square$ stands for concatenation.

Each segment register contains the high-order bits of the starting address of a block of memory of size 2^{13}, just as in the page register example. However, the segment register contains 12 bits rather than 5, and addition rather than concatenation is used to form a physical address so that the starting addresses of segments are not constrained to page boundaries. Thus the PDP-11 segmentation scheme has some additional flexibility over the paging scheme described earlier. There are other facets of the PDP-11 segmentation scheme, not described here, that permit segments to be variable in size up to size 2^{13} and that permit other more general behavior. The important notion is that this scheme uses a combination of concatenation and addition to form a physical memory address and thus combines some aspects of paging with some aspects of base registers.

We emphasize in closing that paging can be used in conjunction with other address modification mechanisms. Indeed this is just what happens in many models of System/370 computers. These computers perform address calculations identical to those of their System/360 predecessors and then modify the effective addresses through a paging operation similar to that shown in Figure 4.4. The reasons for using both paging and base registers stem from the need to fit several programs into memory concurrently and to permit programs or portions of programs to be moved from place to place in memory during program execution.

This type of addressing increases the flexibility and improves the utilization of the computer system. Quite obviously, the intent of such a system is to permit several programs to share a computer system concurrently and to permit a program to be suspended, moved to an auxiliary memory, and then returned to a different place in main memory. When several programs share a computer concurrently, as they can when paging is available, the computer is said to be *multiprogrammed*. Multiprogramming permits efficient utilization of computer time, since a program that must wait for some reason, perhaps for an input from a teletypewriter, can be suspended and a different program can be allowed to resume execution.

4.2 LOGIC OPERATIONS

We have come to adopt the view that each word in a computer memory holds a single symbol. If the computer word contains n bits, then the alphabet of symbols that can be stored in a computer word has 2^n different symbols. We can adopt another view, however, in which each computer word is a collection of n symbols and each symbol is drawn from the alphabet containing only 0 and 1. Each of these views is compatible with the operation of the computer, and each is useful in particular applications. Since it is usually necessary to represent a very large collection of distinct numbers, it is usually necessary to store one number per computer word. When the information stored in memory is a set of answers from a true-false quiz, then n answers per word can be stored by letting each bit represent the answer to a single true-false question. These two views are, moreover, not the only ones possible. A computer word can be partitioned into several subsets of bits such that each subset has its own interpretation. The mantissa and exponent of floating-point representation fall into this category.

Two advantages accrue from adopting a bit-oriented view of the contents of computer words instead of the one-symbol-per-word view. The first advantage is that memory can be used with great efficiency if many items are stored per word instead of just one item per word. Somewhat less important is the fact that parallel computation is possible because there are instructions that operate on all the bits of a word. When these bits are interpreted to represent many different symbols, we have a case in which several symbols can be manipulated simultaneously.

Among the bit-oriented instructions are the logical instructions, which include the AND, OR, EXCLUSIVE OR, and the NOT operations. During execution of instructions from this group the several bits of a single computer word are manipulated identically and independently. The AND, OR, and EXCLUSIVE OR instructions each operate on a pair of oper-

ands, where the *i*th bits of each operand are combined to give the *i*th bit of the result. The following table gives the functional behavior of these instructions:

FIRST OPERAND	SECOND OPERAND	AND	OR	EXCLUSIVE OR
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Note that the AND operation yields a 1 result if both the first *and* the second operands are 1. The OR operation gives a 1 if either the first *or* the second operand is 1; the EXCLUSIVE OR *excludes* the case in which both operands are 1 from those conditions that yield a 1 output for the OR operation.

The NOT operation complements the bits of a computer word. As noted earlier, the COM instruction performs this operation in the PDP-11 computer. The following table describes the logical instructions as they exist on the PDP-11 computer.

INSTRUCTION	CODE	EXAMPLE	ACTION
AND (Bit Test)	BIT	BIT X,AC	Compute AC AND MEMORY[...;X], then set the condition codes depending on the result of the operation. The source and destination operands are not altered.
OR (Bit Set)	BIS	BIS X,%2	REG[2] ← REG[2] OR MEMORY[...;X];
(Bit Clear)	BIC	BIC X,%4	REG[4] ← (NOT MEMORY[...;X]) AND REG[4];
NOT (Complement)	COM	COM %3	REG[3] ← NOT REG[3];

The PDP-11 does not have an instruction that leaves the AND of two operands in memory. An AND can be simulated by the instruction sequence:

```
COM X
BIC X,AC
```

EXAMPLE 4.9

Let the AC contain the 16-bit quantity $A73E_{16}$ and let memory locations X and Y contain $3BB6_{16}$ and 7111_{16}, respectively. The successive states of the AC after executing the instructions listed in the table above become:

INSTRUCTION	RESULT
COM X	$C449_{16}$
BIC X,AC	2336_{16}
BIS Y,AC	7337_{16}
COM AC	$8CC8_{16}$

EXAMPLE 4.10

The PDP-11 does not have an instruction to perform the EXCLUSIVE OR in one operation. The following sequence of instructions forms the EXCLUSIVE OR of the contents of the AC and memory location X and places the result in the AC according to the formula

AC EXCLUSIVE OR X = [(NOT X) AND AC] OR [(NOT AC) AND X]:

CODE	OPERAND	COMMENTS
MOV	AC,%1	Copy AC.
BIC	X,AC	Form (NOT MEMORY[. . . ;X]) AND AC.
MOV	X,%2	Move MEMORY[. . . ;X] to REG[2].
BIC	%1,%2	Form (NOT AC) AND MEMORY[. . . ;X]
BIS	%2,AC	Form the OR of the partial results

System/360-370 computers have full-word logical instructions that are quite similar to those of the PDP-11 computer. These instructions are as follows:

INSTRUCTION	CODE	EXAMPLE	ACTION
AND	N	N 5,T	REG[5] ← REG[5] AND MEMORY[T; . . .];
AND (register)	NR	NR 5,6	REG[5] ← REG[5] AND REG[6];

INSTRUCTION	CODE	EXAMPLE	ACTION
OR	O	O 6,V	REG[6] ← REG[6] OR MEMORY[V;...];
OR (register)	OR	OR 6,8	REG[6] ← REG[6] OR REG[8];
EXCLUSIVE OR	X	X 3,W	REG[3] ← REG[3] EXCLUSIVE OR MEMORY[W;...];
EXCLUSIVE OR (register)	XR	XR 3,4	REG[3] ← REG[3] EXCLUSIVE OR REG[4];

There is no equivalent of the COM instruction to form the logical complement of an operand. The complement can be computed, however, by using the X instruction with an operand equal to

$111\cdots1_2$ = FFFF $FFFF_{16}$

EXAMPLE 4.11

Let REG[2] contain 0000 $A73E_{16}$ and let the full words in memory with addresses X, Y, Z, W contain 0000 $3BB6_{16}$, 0000 7111_{16}, 0000 $9D31_{16}$, and FFFF $FFFF_{16}$, respectively. Compare the following instructions with those in Example 4.9:

INSTRUCTION	RESULT
N 2,X	0000 2336_{16}
O 2,Y	0000 7337_{16}
X 2,W	FFFF $8CC8_{16}$
X, 2,Z	FFFF $11F9_{16}$

4.3 SHIFT OPERATIONS

We noted the need for aligning operands in the discussion on arithmetic operations; by *alignment* we mean the process of shifting bits left or right in a computer word. The operation is a simple one conceptually, but there are a number of variations that bear discussion.

Figure 4.6 diagrams several different operations that are commonly used. The cyclical shift (sometimes called *rotate*) is the operation in which bits shifted off one end of a register reappear at the other end. Although the figure shows a right cyclical shift of one bit, the effect is exactly the same as a left cyclical shift of 15 bits. A left cyclical shift of m bits can

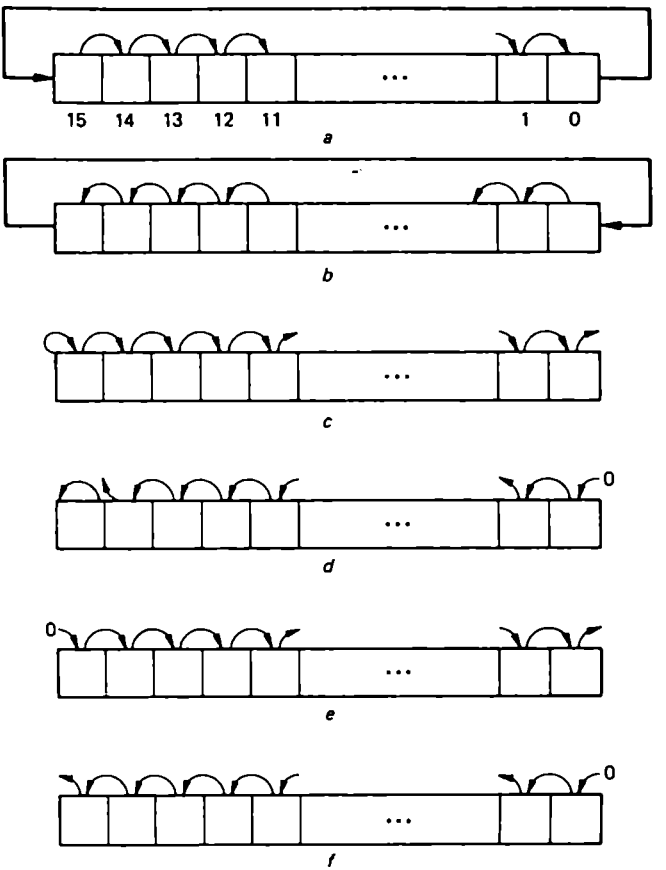


FIGURE 4.6 Shift operations: (a) right cyclic shift, (b) left cyclic shift, (c) arithmetic right shift, (d) arithmetic left shift, (e) logical right shift, and (f) logical left shift

always be made as a right cyclical shift of $n - m$ bits in a computer with n bits per word. Consequently, instruction repertoires often do not include separate instructions for both left and right cyclical shifts.

The arithmetic right shift and arithmetic left shift shown in Figure 4.6 are shifts that leave the sign bit of an arithmetic operand unchanged. For two's-complement representation the arithmetic left shift is the same as a multiplication by 2 and the arithmetic right shift is the same as an integer division by 2. Bits that are shifted off the end of the register are lost. Note that the cyclical shift can be used to double and halve numbers in ones'-complement representation, provided that the sign bit remains unchanged. What shift instructions double and halve numbers in signed-magnitude representations?

Logical shifts are useful primarily for isolating fields within a word.

The 0's that are shifted into a word displace unwanted information, which is shifted off the end of the word.

In general, instruction repertoires do not include all the shifts depicted in Figure 4.6. However, they do include enough shift instructions that all the variations can be programmed. This is true of both the PDP-11 and the System/360-370.

The PDP-11 shift instructions are:

INSTRUCTION	CODE	EXAMPLE	ACTION
Rotate right	ROR	ROR AC	Right cyclic shift REG[AC] one place with condition code register C.
Rotate left	ROL	ROL %1	Left cyclic shift REG[1] one place with condition code register C.
Arithmetic shift right	ASR	ASR X	Arithmetic right shift REG[X] one bit with condition code register C (C receives bit 0)
Arithmetic shift left	ASL	ASL %4	Logical left shift REG[4] with condition code register C (C receives bit 15).

The carry bit participates in all the shift and rotate operations as bit number 16. Figure 4.7 illustrates these PDP-11 instructions. Condition code register C is shown as an extension to a 16-bit word in the figure.

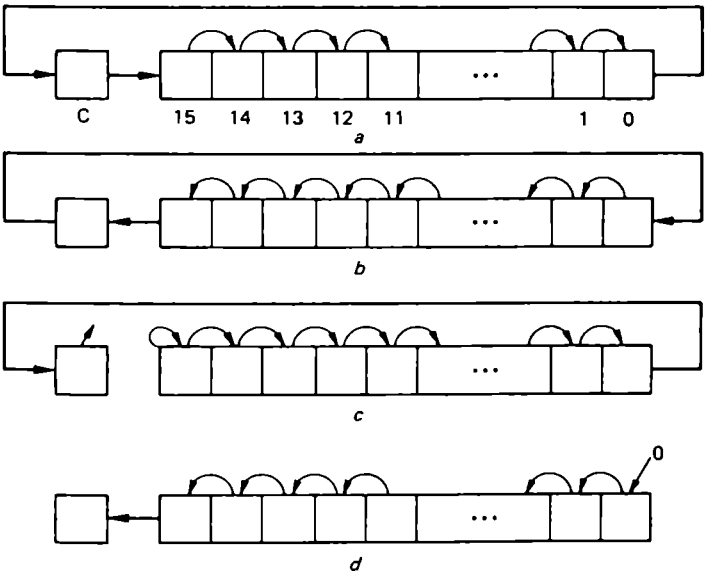


FIGURE 4.7 The PDP-11 shift operations: (*a*) rotate right (code ROR), (*b*) rotate left (ROL), (c) arithmetic shift right (ASR), and (*d*) arithmetic shift left (ASL)

To simulate the various shifts described in Figure 4.6, the effect of the carry bit has to be neutralized. The right cyclic shift can be implemented as shown in Example 4.12.

EXAMPLE 4.12

The instructions below perform a right cyclic shift of an operand in register AC.

ADDRESS	CODE	OPERAND	COMMENTS
	CLR	%1	Clear REG[1] and C condition code.
	ROR	AC	Rotate AC right. The original bit 0 of AC appears in the C condition code.
	ROR	%1	The original bit 0 of AC is now bit 15 of REG[1].
	BIS	%1,AC	OR the registers. Bit 15 of AC now contains the original value of bit 0.

The ASL instruction in the PDP-11 is not a true arithmetic left shift as depicted in Figure 4.6 in that the sign bit is replaced by bit 14 rather than by itself. The arithmetic left shift can be simulated by the following set of instructions. It is sometimes useful, as in Example 4.13, to set or clear a condition code register. Accordingly, the PDP-11 instruction repertoire has four condition-code-setting instructions (codes SEC, SEZ, SEN, and SEV) for setting the four condition code registers C, Z, N, and Z, respectively. Similarly, there are four condition-code-clearing instructions, CLC, CLZ, CLN, and CLV.

EXAMPLE 4.13

This instruction sequence does an arithmetic left shift of an operand in register AC. Assume that SIGN is the symbolic addresses of a memory cell that contains 100000_8.

ADDRESS	CODE	OPERAND	COMMENTS
	ASL	AC	Shift.
	BVC	EXIT	Branch if sign bit did not change.
	ADD	SIGN,AC	Complement the sign bit, restoring to its original value.
	SEV		Overflow has occurred, set overflow condition code.
EXIT:			

The ASR instruction performs a true arithmetic right shift, and the ASL instruction can be used for a logical left shift. How do you perform a left cyclic shift or a logical shift of a 16-bit word? Each can be done in two instructions.

Occasionally a single computer word is not big enough to hold a single datum, in which case the datum is stored in two adjacent memory words. When two words are used to hold one datum, we say that the datum is a *double-precision* operand. Example 4.14 illustrates a PDP-11 instruction sequence for the rotation of a double-precision datum. The add carry (code ADC) instruction forms the sum of the operand and the carry condition code and replaces the operand by the sum. The ADC instruction is particularly useful in programming double-precision operations.

EXAMPLE 4.14

Assume a double-precision datum is stored in memory locations X and Y (X contains the more significant portion of the datum). A left rotate is to be performed. That is, bit 15 of Y is to be rotated into bit 0 of X and bit 15 of X into bit 0 of Y, with both X and Y shifted left.

ADDRESS	CODE	OPERAND	COMMENTS
	ASL	Y	Shift Y. The carry condition code now contains bit 15 of Y.
	ROL	X	Rotate X left. The carry condition code now contains bit 15 of X, and bit 0 of X contains the original bit 15 of Y.
	ADC	Y	Bit 0 of Y now contains the original bit 15 of X.

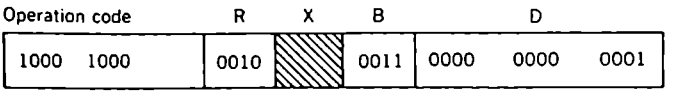


FIGURE 4.8 The encoding for the logical right shift instruction SRL 2,1(3). The X field is not used for this instruction

System/360-370 computers have arithmetic and logical shifts, as depicted in Figure 4.6, but no cyclical shifts. The shift instructions have an interesting property that is not available for the PDP-11 computer. They use a format similar to the RX format, but the memory address is interpreted as a shift amount. Hence one instruction can be used to shift by any amount. Moreover, the shift amount is computed from the B and the D fields as if it were an effective address, so that shift amounts can be computed dynamically.

To be more explicit, Figure 4.8 shows the format for the logical right shift instruction, SRL. The shift amount for the instruction is determined by extracting the least significant six bits from the quantity REG[B]+D. Register R is shifted when the instruction is executed; and if the shift amount is greater than 32, the 32-bit register will be cleared to 0 by the logical shift. There are double-register shift instructions as well as single-register instructions, all of which are given in the following table:

INSTRUCTION	CODE	EXAMPLE	ACTION
Shift right	SRL	SRL 2,1	Logical right shift of REG[2], one bit.
Shift right (double)	SRDL	SRDL 2,1	Logical right shift of REG[2;3], one bit.
Shift left	SLL	SLL 3,17	Logical left shift of REG[3], 17 bits.
Shift left (double)	SLDL	SLDL 4,6	Logical left shift of REG[4;5], six bits.
Shift right arithmetic	SRA	SRA 2,1	Arithmetic right shift of REG[2], one bit.
Shift right arithmetic (double)	SRDA	SRDA 2,1	Arithmetic right shift of REG[2;3], one bit.
Shift left arithmetic	SLA	SLA 5,7	Arithmetic left shift of REG[5], seven bits.
Shift left arithmetic (double)	SLDA	SLDA 6,2	Arithmetic left shift of REG[6;7], two bits.

EXAMPLE 4.15

The right cyclic shift of a datum in REG[2] can be programmed on the System/360-370 as follows. The SRDL instruction is used to perform the logical shift and to capture the least significant bit of the datum in REG[3].

CODE	OPERAND	COMMENTS
SR	3,3	Clear REG[3].
SRDL	2,1	Shift REG[2] right one bit. The most significant bit of REG[3] now contains the least significant bit of the operand formerly in REG[2].
OR	2,3	The most significant bit of REG[2] is replaced by the least significant bit of its former operand.

EXAMPLE 4.16

X, Y, and Z are the addresses of full words in memory. The following instructions leave Y ÷ Z in REG[7]. The SRDA instruction is used to extend the sign of the dividend through REG[6], producing a double-word representation of the full-word operand.

CODE	OPERAND	COMMENTS
L	6,Y	REG[6] ← MEMORY[Y; . . .].
SRDA	6,32	Shift REG[6;7]. After shift, REG[7] contains previous contents of REG[6], and sign of operand has been extended through REG[6].
D	6,Z	REG[7] ← Y÷Z. REG[6] ← remainder.

EXAMPLE 4.17

Let Y be the address of a full word in memory that contains a positive integer less than 32. The following instructions shift REG[3] left arithmetically by MEMORY[Y; . . .]+1 places. Since the SLA instruction has no X field, we drop the X field from the operand address notation.

CODE	OPERAND	COMMENTS
L	4,Y	REG[4] ← MEMORY[Y; . . .].
SLA	3,1(4)	Shift REG[3] left (REG[4]+1) places.

In the next section we conclude the current discussion by developing a general classification of computers as to the number of addresses per instruction.

4.4 CLASSIFICATION OF COMPUTERS

A significant amount of information about a computer can be conveyed by specifying the number of addresses contained in an instruction. Indeed, in the past it was quite fashionable to classify computers in this manner. This method of classification is seldom used today because most modern computers have a large number of addressing modes and cannot be classified into one category. In this section we examine the various computer classifications and also give examples of PDP-11 and System/360-370 addressing modes when they are appropriate.

4.4.1 Three-Address Computers

When a processor executes an instruction, there are at least five pieces of information it must have to complete the execution:

1. The type of operation to be performed
2. The address of the first operand
3. The address of the second operand (if any)
4. The address of the result
5. The address of the next instruction

A three-address computer instruction contains four of these items explicitly, namely, an operation code and three addresses (one each for source operand one, source operand two, and the destination). Symbolically, a three-address instruction is represented as:

```
SUB S1,S2,D
```

The effect of the instruction is to subtract the operand at source location S2 from that at source location S1 and place the result in destination lo-

cation D. A program example for a three-address computer appears in Example 4.18.

EXAMPLE 4.18

The following program computes X = (A*B/C)−D with three-address instructions:

LABEL	CODE	OPERAND	COMMENTS
	MULT	A,B,TEMP1	Multiply A and B and place the result in TEMP1.
	DIV	TEMP1,C,TEMP2	Divide A * B by C and place the result in TEMP2.
	SUB	TEMP2,D,X	Subtract D from TEMP2 and place the result in X.

4.4.2 Two-Address Computers

The major drawback with three-address computers is that every instruction has to have enough bits to represent three addresses. Since memory is often one of the dominant costs of a computer, designers strive to keep the number of bits per word as small as possible. An obvious reduction of instruction size is to reduce the number of addresses from three to two. Note that if the operand B in Example 4.18 is required to form only the product A*B, then the destination address could be B, as in MULT A,B,B. In two-address computers the destination address is implicitly assumed to be the same as that of one of the source operands. For System/360-370 computers, the result register is the same as the first operand. For the PDP-11, the result is the same as the second operand. Following the PDP-11 convention, we can rewrite Example 4.18 for a two-address machine as follows:

```
MULT    A,B
DIV     B,C
SUB     C,D
MOV     D,X
```

Note that the two-address computer requires one more instruction than the three-address computer for this example, but programs for the two-address computer can usually be encoded in fewer bits than programs for three-address computers.

4.4.3 One-Address Computers

A further reduction in the number of bits is achieved by including only one address per instruction. Now, an implicit address is required not only for the destination but also for the second source operand. This is usually obtained by assuming that an internal machine register, an accumulator, is the source of the second operand as well as the destination. Example 4.19 depicts the calculation X = (A*B/C)—D written for a typical one-address computer. Again note that the one-address computer requires more instructions to perform the same calculation. It is true in general that the fewer addresses contained in an instruction the longer an instruction sequence must be to perform a given calculation; but since fewer bits per instruction are required, the total number of bits to encode a program may not be significantly different.

EXAMPLE 4.19

This shows the calculation of X = (A*B/C)—D for a one-address computer.

LABEL	CODE	OPERAND	COMMENTS
	LOAD	B	Load the accumulator with B.
	MULT	A	Multiply the contents of the accumulator by A and place the results in the accumulator.
	DIV	C	Divide the accumulator by C.
	SUB	D	Subtract D from the accumulator.
	STORE	X	Store the accumulator in X.

4.4.4 Plus-One-Address Computers

Both the PDP-11 and the System/360-370 computers use the contents of an internal register, the P register (or program counter), to specify the address of the next instruction. Most instructions increment the P register by their own lengths, so that the P register contains the address of the immediately following instruction at the completion of execution of the present instruction. Such instructions contain no explicit reference to the next instruction. Branch instructions modify the P register to indicate the address of the branch. These instructions contain explicit infor-

mation concerning how to compute the address of the next instruction. For example, conditional branch instructions specify that the P register should be changed to a branch address if certain conditions are satisfied; otherwise, the P register should be incremented to indicate the next consecutive instruction. This type of computation is typical of most modern computers of the 1970s, but computers of the early 1950s used another method to specify the next instruction. It differed from the present method primarily because of the change in memory technology.

Modern memory technology is based upon random-access storage. In this type of storage, the time required to access an item is constant and independent of previous accesses to memory. In the early 1950s main memories were rotating magnetic drum memories rather than random-access memories. A magnetic drum memory has, in a fixed position, a read/write head that reads information from a rotating drum as it revolves under the head. If two items are adjacent on the drum surface, then the second one can be read immediately after the first with no latency time between. To read the items in reverse order, the second one is read and then no further access is made while the drum rotates almost an entire revolution until the first one reaches the read head. A typical drum had room for 100 to 200 items around its circumference, and thus the effect of a full revolution of drum latency could increase access time to 100 to 200 times that of an access made without latency. Thus there is a premium for reducing latency with such memories.

For this reason, instructions for cyclic memory computers indicate the address of the next instruction explicitly. Thus a clever programmer can place instructions at specific locations around the drum, so that each instruction specifies its successor to be the instruction that just comes under the read head when execution of the present instruction is completed. The programmer, of course, must take into account the location of data on the drum and the time required to execute various instructions. To indicate the address of the next instruction, each instruction contains an extra address field in addition to the normal address fields. Any one of the three previously described classifications (one-, two-, and three-address) can have the address of the next instruction included in its format, which yields formats called *one-plus-one address*, *two-plus-one address*, and *three-plus-one address*. The generic term applied to formats that specify the next instruction address explicitly is thus *plus-one-address*.

As random-access memories came into wide use, drum memories rapidly fell into disfavor for use as main memories, and with that change the plus-one-address format fell into disuse. Of late, however, advanced memory technologies have produced very small and inexpensive shift register memories that behave functionally much like drums, although

there is no mechanical rotation. It is possible that the plus-one-address format may be resurrected if these become widely available for use as main memories.

4.4.5 Comparison of Addressing Schemes

Before closing this chapter it is worthwhile to compare the various addressing schemes mentioned here. The cost of the addressing schemes varies in two ways, namely, the number of bits required to encode an algorithm and the time required to execute an algorithm. As a general rule, execution time decreases as the number of instructions decreases, and this suggests that three-address instructions have the advantage for minimizing execution time. On the other hand, three-address instructions waste memory space when an operand address is the same as a destination address because the instruction ADD X,Y,Y in a three-address form has one redundant address. The two-address version of this instruction is ADD X,Y, and it does the same operation as the three-address version but with fewer bits required to specify the operation. Using this reasoning, we see that there is some advantage to instruction formats in which address fields rarely have useless information.

Three-address instruction repertoires have rarely been used in computers, probably because of the wastefulness of the repertoire in encoding programs. When they have been used, they have been used for specific reasons. The CDC 6600 computer is one example of a computer with a three-address instruction set, and here the instruction format permits the computer to perform some instructions simultaneously or out of sequence if the execution time can be reduced without destroying the correctness of a program. A two-address instruction repertoire does not lend itself as well to such an application, although the IBM 360/91 does attempt to process two-address instructions in a manner somewhat similar to the CDC 6600 processing methodology.

The one-address instruction set has fallen into disfavor because the single-accumulator computer has given way to the multiaccumulator computer in all but the smallest and least expensive computers. Naturally, multiaccumulator computers cannot have one-address arithmetic instructions.

An interesting type of instruction not described in this chapter is the *zero-address* instruction, and, indeed, a few computers have been constructed with such instruction repertoires. The discussion of this repertoire is postponed until Chapter 10. In reality, no computer has a repertoire consisting entirely of instructions with zero addresses. Some instructions must have a single address. Nevertheless, the instruction encodings for this type of machine can be very dense, because many

operations are encoded implicitly in this type of machine whereas in other computers the operation must be encoded explicitly.

Current trends in computer design have led to large instruction repertoires with many instruction formats in a single machine. Often, instructions with one, two, and three addresses are found together in a computer. The trend in general-purpose computers is toward flexibility through large numbers of instructions of varying formats. Memory lost through wasteful encodings is not a major issue in this class of machines. For minicomputers the trend is not so clear because memory is a precious resource in such machines. Clever encodings of instructions are clearly beneficial. The PDP-11 addressing certainly shows one way of approaching the problem. Frequently used addresses are registers and require only three bits to specify. Less frequently used addresses are memory addresses, which require three instruction bits plus one 16-bit address word. Thus there is a substantial memory saving in a program if most of the operations are register operations.

We return to the description of instruction formats in Chapter 10, where we investigate zero-address computers.

REFERENCES

For a detailed discussion of computer classification and a comparison of instruction encoding efficiencies see Chapter 3 of Bell and Newell (1971). Chapter 39 of the same reference contains a description of the CDC 6600. The January 1967 issue of *IBM Journal of Research and Development*, vol. 11, no. 1, is devoted to the IBM 360/91.

EXERCISES

4.1 Invent a method for using only absolute addressing and indirection in an instruction repertoire. Describe the registers that take part in these operations, the encoding of instructions for specifying the operations, and the sequence of events that occur when an instruction is executed. Take advantage of any economies, such as a smaller instruction encoding, stemming from the simple addressing modes.

4.2 Show how displacements can be implemented in the PDP-11 instruction repertoire as an effective address calculation in addition to the register and relative addressing modes.

4.3 Write a sequence of instructions that adds two full-word operands, using only the logical and shift instructions to perform the processing. You may use the BCT instruction for System/360-370 or the INC instruction for the PDP-11, but the number of iterations counted by these instructions must not exceed the number of bits per word.

4.4 Write a sequence of instructions that multiplies two operands. The sequence may include logical, addition, and shift instructions, but not a multiplication instruction. Assume that the operands are positive integers less than $256 = 2^8$. You may use either the BCT or the INC instruction to control the iterations of a loop, but the number of iterations must not exceed 8.

4.5 Write a simulator for the PDP-11 in a high-level language such as ALGOL or FORTRAN. The simulator takes a series of 16-bit strings as input. These strings represent the binary encoding of PDP-11 instructions, address words, and data. The simulator fetches instructions from an array, performs the necessary address calculations to obtain source and destination operands, and modifies the operands according to the operation code. The ALGOL-like descriptions in Sections 2.7 and 4.1.4 can be used as the basis for a simulator.

FIVE

ASSEMBLERS

In the stored-program computer, instructions are held in memory in a binary encoding. We can, in principle, put a program into a computer by placing each instruction of the program in encoded form at its proper place in memory, one instruction at a time from the console of the computer. The weakness of this method becomes evident to anyone who tries it. Our program examples have stressed the use of symbolic addresses and mnemonic instruction codes because these quantities are much easier to use than binary codes. Since instructions can be mechanically translated from symbolic form into a binary encoding, it is obvious that the translation process can be performed by a computer program. Consequently, programs called *assemblers*, which perform these translations, have been written for nearly every computer. This chapter discusses assembly language as a programming language and describes the translation process as it is performed by assemblers.

5.1 ASSEMBLY LANGUAGE

The assembly process is one in which the binary encoding for each portion of an instruction is determined and the several encodings are then combined to form a complete instruction. For example, the instruction INC X, where X is a symbol representing an address 70_8 words beyond the address word of the current instruction, is assembled by combining the code INC with the encoding for the relative addressing mode. The two encodings are positioned in the correct places in a 16-bit computer word, and that computer word is filed away. The address word is assembled with 66_8 and also filed away. (Recall that the offset is calculated as X−(P+2), where P is the address of the address word.) When an assembled program is put in the computer for execution, another program, called a *loader*, places each assembled instruction in the proper memory location for execution.

More specifically, an assembler for PDP-11 computer instructions must determine the following quantities for each instruction:

1. The address at which each instruction is to be assembled
2. The instruction code
3. The source address field
4. The destination address field
5. An additional address word for the source operand, if necessary
6. An additional address word for the destination operand, if necessary

The assembler determines these quantities from information in an assembly language program.

Assembly language for the PDP-11 computer has the following structure. Each instruction is described by one line of text, and each line has four fields named label, code, operand, and comments. The label field is often blank. If it is not blank, the symbol in that field represents the instruction address and may be used as a synonym for that location throughout the assembly language program. The code field contains mnemonics for the operation codes, and the operand field contains source and destination operands. Descriptive remarks are placed in the comments field and separated from the other fields by a semicolon. The comment field is ignored by the assembler. Many assembly-language-like examples have already appeared in the text.

In addition to mnemonics for instruction codes, assembly language

contains instructions that control the assembly process and do not have equivalent machine instruction codes. These codes are called *pseudo-instructions.* An important example of a pseudo-instruction for the PDP-11 computer is the end instruction (code .END), which specifies that the end of the program has been reached. Another type of pseudo-instruction, called a *direct assignment* and identified by the operator =, assigns a numeric value to a symbolic name that appears in the program. In assembly language the symbol ., called the *program location counter* (PLC), denotes the address in memory at which the item will be assembled. The dot in any instruction of a program denotes the address of the first word assembled for that instruction. The use of the program location counter and the assembler pseudo-instructions should become clear by inspection of a sample program.

Example 5.1 shows the use of the program location counter, direct-assignment, and .END pseudo-instructions in an assembly language program. In this case, X, Y, and Z are symbols that indicate memory addresses 005126_8, 000300_8, and 005130_8, respectively. The codes for MOV, CMP, BEQ, ADD, and BR are 0001_2, 0010_2, 00000011_2, 0110_2, and 00000001_2,

EXAMPLE 5.1

```
     LABEL   CODE    OPERAND      COMMENTS
1.   .       =       5000         ; Start assembly at 5000₈.
2.   AC      =       %0           ; Equate symbol AC with register
                                  ; zero.
3.           MOV     X,AC         ; Fetch X.
4.           CMP     AC,Z         ; Compare AC to Z.
5.           BEQ     MOVE         ; Branch if X = Z.
6.           MOV     Y,AC         ; Fetch Y.
7.           ADD     Z,AC         ; Add Z.
8.           MOV     AC,X         ; Store result in X.
9.   HLT:    HALT                 ; Stop.
10.  MOVE:   MOV     AC,Y         ; Store X in Y.
11.          BR      HLT
12.  X       =       5126         ; X is address 5126₈.
13.  Y       =       300          ; Y is address 300₈.
14.  Z       =       5130         ; Z is address 5130₈.
15.          .END
```

EXAMPLE 5.1 *(Continued)*

PROGRAM LOCATION COUNTER	OP CODE	SOURCE	DESTINATION	OFFSET	ASSEMBLED INSTRUCTION	SYMBOLIC INSTRUCTION
...	...	...	...	...	None	
...	...	...	...	...	None	
5000	0001	110 111	000 000	–	016700_8	MOV –,AC
5002	–	–	–	–	000122_8	X
5004	0010	000 000	110 111	–	020067_8	CMP AC,–
5006	–	–	–	–	000120_8	Z
5010	0 000 001 1	–	–	00 000 111	001407_8	BEQ MOVE
5012	0001	110 111	000 000	–	016700_8	MOV –,AC
5014	–	–	–	–	173262_8	Y
5016	0110	110 111	000 000	–	066700_8	ADD –,AC
5020	–	–	–	–	000106_8	Z
5022	0001	000 000	110 111	–	010067_8	MOV AC,–
5024	–	–	–	–	000100_8	X
5026	0000	000 000	000 000	–	000000_8	HALT
5030	0001	000 000	110 111	–	010067_8	MOV AC,–
5032	–	–	–	–	173244_8	Y
5034	0 000 000 1	–	–	11 111 100	000774_8	BR HALT
...	...				...	

respectively. HALT is encoded as the 16-bit instruction 000000_8. When the HALT instruction is executed, the computer does not return to the fetch state to process further instructions.

Several important points are illustrated in Example 5.1. The pseudo-instruction . = 5000 forces assembly to begin at location 5000_8. All integers are treated as radix-8 integers unless they contain a decimal point, in which case they are treated as radix-10 integers. The direct assignment in line 2 establishes that the symbol AC denotes register zero. The symbol % notifies the assembler that a register is being referenced. The symbols X, Y, and Z are given numeric values by direct-assignment statements in lines 12 through 14.

Before we describe the pseudo-instructions in the assembly language for the PDP-11, it is necessary to consider a few details of the assembly process. The complete assembly process is described later. One of the essential tasks of the assembler is to replace symbolic names for memory addresses by their internal encodings. In order to perform this task, the assembler constructs a table of symbolic names called the *symbol table.* In the symbol table each symbolic name is stored in the appropriate character representation, together with the memory address that it represents. Thus the symbol table for the program in Example 5.1 is as follows:

SYMBOL	ADDRESS
HLT	5026_8
MOVE	5030_8
X	5126_8
Y	0300_8
Z	5130_8
AC	REG[0]

Consider how the assembler builds a symbol table. In general, there are two ways in which a symbol is assigned a value: (1) by appearing in the label field of a line of text, in which case its value is the value of the memory address represented by that line of text, and (2) by appearing on the left-hand side of a direct-assignment statement.

In order to determine the value of a label when it is encountered, the assembler must keep track of the current value of the program location counter. Example 5.1 shows that the assembler can compute the addresses for the labels HLT and MOVE by counting from 5000_8, incrementing the program location counter by 2, 4, or 6 for each instruction depending on whether zero, one, or two address words are associated with

the instruction. Hence when an assembler processes Example 5.1, the value of the program location counter is 5026_8 at the point where HLT appears.

The assembler must be able to determine the contents of the source and/or destination fields. For each of the addressing modes described in Chapter 4 there is a corresponding symbolic representation that the assembler recognizes. This correspondence is shown in Table 4.1. Recall that the relative, relative deferred, index, and index deferred addressing modes utilize address words immediately following the instruction in the effective-address calculation. Another task of the assembler is to calculate and assemble the addresses in these words.

For an illustration of address-word processing, reconsider Example 5.1. The MOV instruction in line 3 contains the symbol X as a relative address. The address word for this reference is stored at address 5002, and the address of X is 5126_8 because of the direct assignment in line 12. Recall that the P register is incremented immediately after the fetch of a word from memory so that it always contains the address of the next word to be fetched. Thus after the word at location 5002_8 is fetched, the P register will contain 5004_8. If b is the contents of location 5002, b + 5004 = X = 5126. The value assembled at location 5002 is thus $5126_8 - 5004_8 = 0122_8$. Likewise, $000\ 120_8$ is assembled at location 5006_8 for the CMP AC,Z instruction. The value assembled for location 5014_8 is negative, since the symbol Y represents an address that precedes the address word for this reference to Y.

The offsets for branch instructions are computed in a similar manner. Since branches can be made only to other instructions and instructions occupy multiples of full words, the offset in a branch is actually calculated in words rather than in bytes. Thus the BEQ instruction assembled at location 5010_8 has an offset calculated as [MOVE − (5010 + 2)]/2. Note that negative offsets result in backward branches such as in the BR instruction assembled at location 5034_8. Here the leading bit of the 8-bit offset is the sign bit, and two's-complement representation is used.

In summary, the normal action of the assembler in processing an instruction is as follows:

1. If there is a symbol in the label field, place the symbol and the value of the program location counter in the symbol table.
2. Determine the instruction type. Assemble the instruction code.
3. Determine the source and destination operands of the instruction (if any) and assemble them in the source and destination fields, respectively.

4. If no source or destination address words are required, then increase the program location counter by 2. If the instruction requires a source or a destination address word but not both, assemble it and increase the program location counter by 4. Otherwise, if the instruction requires both a source and destination address word, assemble them and increase the program location counter by 6.

A brief description of the pseudo-instructions used in Example 5.1 and of other important ones in the assembly language for the PDP-11 computer appears in Table 5.1. We show how these are used in Example 5.2. Note that we use the abbreviation PLC for program location counter and that the phrase *enter* (R,PLC) *in symbol table* means that the symbol R and the current value of the program location counter are entered as a symbol and its corresponding memory address.

Example 5.2 illustrates the use of these pseudo-instructions. Recall that the origin pseudo-instruction (. =) initializes the program location counter. Hence the first line in Example 5.2 instructs the assembler to start assembly at 5126_8.

The direct-assignment pseudo-instruction causes the assembler to enter a symbol and its address into the symbol table without changing the program location counter. In Example 5.2 there is a direct-assignment instruction that causes X to be placed in the symbol table. Note carefully that the program location counter is not changed by this instruction, so that Y is assembled at 5126_8, the same address at which it would be assembled if the direct-assignment instruction were omitted.

The direct assignment in the second line is used for two reasons. One reason for the direct-assignment pseudo-instruction is that the symbolic name AC is more descriptive and informative than %0. The program in the example would be assembled identically if AC were replaced by %0 in every reference. A second reason for using the direct-assignment instruction in this manner is more subtle. We can change the register named AC by changing the definition of AC in the direct-assignment pseudo-instruction. In effect, one change by the programmer can be propagated through the program. If, however, we used %0 in place of the symbol AC, then we would have to change several instructions in order to change the register to some other register. Moreover, in a large program there may be many references to REG[0], not all of which pertain to it in the same context. Reassignment of REG[0] in the AC context in such a program would require careful examination of every reference to REG[0]. Clearly, the direct-assignment instruction greatly assists in the task of reassigning registers when such changes must be made.

The . = . + instruction is used to reserve one or more bytes of mem-

TABLE 5.1

PSEUDO-INSTRUCTION	CODE	EXAMPLE	ASSEMBLER ACTION
Origin	. =	. = 1000	PLC ← 1000_8.
Direct assignment (equate)	=	X = 300	Enter (X,300_8) in symbol table.
	=	A = 900.	Enter (A,900_{10}) in symbol table.
Block storage	. = . +	R: . = . + 10	Enter (R,PLC) in symbol table. PLC ← PLC+10_8. (This reserves 10_8 bytes.)
Decimal		Z = 100.	Enter (Z,100_{10}) in symbol table.
Constants		Y: 300	Enter (Y,PLC) in symbol table. Place 300_8 in MEMORY[. . . ;Y]. PLC ← PLC+2.
		B: 10.	Enter (B,PLC) in symbol table. Place 10_{10} in MEMORY[. . . ;B]. PLC ← PLC+2.
Even	.EVEN	.EVEN	If PLC is odd, then PLC ← PLC+1.
Word constants	.WORD	.WORD 176,4.0	Place 176_8 in MEMORY[. . . ;PLC]. PLC ← PLC+2. Place 4_{10} in MEMORY[. . . ;PLC]. PLC ← PLC+2.
Byte constants	.BYTE	.BYTE 2.0,1	Place 2_{10} in MEMORY[PLC]. PLC ← PLC+1. Place 1_8 in MEMORY[PLC]. PLC ← PLC+1.
Character convert	.ASCII	T: .ASCII/ABCD/	Enter (T,PLC) in symbol table. Place representations of characters A, B, C, and D in next two words of memory. PLC ← PLC+4.
End of program	.END	.END	End of processing.

ory for data. If the instruction is labeled, then the label is given the address of the first cell reserved as indicated for SAVE3 in Example 5.2. The assembler does not initialize block storage to a known value. A memory cell in a block storage area will contain whatever happened to be in the cell before the program is loaded.

EXAMPLE 5.2

ADDRESS	CONTENTS	LABEL	CODE	OPERAND	COMMENTS
			=	5126	; Assembly begins at ; 5126_8.
		AC	=	%0	; Equate AC with register zero.
		X	=	300	; Place ($X,300_8$) in ; symbol table.
5126_8		Y: .	=	. + 2	; Save one word. ; This is ; MEMORY[. . .;5126_8].
5130_8		Z: .	=	. + 2	; Save one word.
5132_8		SAVE3: .	=	. + 6	; Save three words.
5134_8					
5136_8					
5140_8	000012_8	TEN:		10.	; Assemble 10_{10}.
5142_8	000010_8	EIGHT:		10	; Assemble 10_8.
5144_8	005126_8	YNAME:		Y	; Assemble address of ; Y.
5146_8	042524_8	TEXT:	.ASCII	/TEXT/	; Assemble "TE" and ; "XT" in next two ; words. "TE" is in ; MEMORY[. . . ;5146_8].
5150_8	052130_8				; "XT" is in ; MEMORY[. . . ;5150_8].
5152_8	00002_8	B:	.BYTE	2	; Assemble 2_8 in the ; next byte.
			.EVEN		; Make next assembly ; location a full word.
5154_8	005144_8	YYNAME:		YNAME	; Assemble address of ; YNAME.
			.END		; End of assembly.

The symbol table for this example is:

SYMBOL	ADDRESS
X	0300_8
Y	5126_8
Z	5130_8
SAVE3	5132_8
TEN	5140_8
EIGHT	5142_8
YNAME	5144_8
TEXT	5146_8
B	5152_8
YYNAME	5154_8

Lines labeled TEN and EIGHT of Example 5.2 cause their operands to be converted into a binary representation and placed in memory. The presence of the decimal point specifies that the operand is radix-10 representation, whereas the absence of the decimal point specifies radix-8. The following line reserves a memory word for a datum, but the datum is the address of another memory word. Thus the memory address of the cell labeled Y is placed in the cell labeled YNAME.

The .ASCII instruction reserves n words of storage, where n is the number of words required to store the text occurring between the delimiters "/". The assembler places the encoded form of the characters that appear in the operand field in this storage area. Each character is represented by an eight-bit code, so that two characters can be placed in a single word. In Example 5.2 the codes for "T", "E", and "X" are 124_8, 105_8, and 130_8, respectively, and the data are so assembled that the characters occupy successive bytes. Thus "T" is in byte 5146_8, "E" in byte 5147_8, "X" in byte 5150_8, and "T" in byte 5151_8. The mnemonic .ASCII is drawn from the acronym ASCII, which is the name of a character-encoding scheme used for input/output devices such as a teletype. Note that Example 5.2 shows the contents of each word with the odd byte on the left and the even byte on the right. All full-word constants appear correctly in this format. However, byte data appear reversed. That is, the "E" in byte 5147_8 appears before the "T" in byte 5146_8.

Finally, the .BYTE pseudo-instruction assembles a byte with the value 2_8 in the next byte and advances the program location counter by one byte. The .EVEN pseudo-instruction ensures that the assembly of any following data area starts on a word rather than a byte boundary by increasing the program location counter by 1 if necessary.

The operand of an instruction need not be a single symbol; it may be an expression involving constants and symbols. The allowable form of expressions is quite limited in some versions of the assembly language for the PDP-11 computer; in particular, an operand can be a constant, a symbol, or a sum or difference of constants and symbols. Some instructions, such as .ASCII, do not permit expressions as operands. However, other versions of the PDP-11 assembly language such as Macro-11, as well as most other assemblers, allow a large class of arithmetic expressions to appear in the operand field.

EXAMPLE 5.3

Let X, Y, and Z be symbols for the addresses 224_8, 536_8, and 172_8, respectively. The address computation shows the assembled word when each of the instructions shown is at memory location 76_8, with address word at 100_8.

CODE	OPERAND	EFFECTIVE ADDRESS	ASSEMBLED ADDRESS WORD
CLR	Y − X	312_8	210_8
INC	Z + 2	174_8	72_8
COM	X − 10	214_8	112_8
DEC	Y − 10.	524_8	422_8
CLR	. + 10	106_8	4_8

Recall that the symbol . is used to indicate the current value of the program location counter. Its symbolic value is the address of the first word of the instruction being assembled. The program location counter is not updated during the assembly of multiword instructions. Consider the instruction CLR . + 10 above. The operand requires an address word, so the instruction occupies two words. The . refers to the address of the first word of the instruction and not to the address word. The instruction BR . + 4 branches over one word because the address of the word immediately after the instruction is . + 2 and the address of the word beyond that is . + 4. The instruction BR . + 2 branches to the immediately succeeding instruction in memory and thus is a null instruction because it does no computation and does not change the flow of instruction sequencing.

5.2 ASSEMBLY LANGUAGE FOR SYSTEM/360-370

The assembly language for System/360-370 computers is very similar to that for the PDP-11 computer. Instructions follow the same format of symbol, operation code, operand, and comments. Moreover, System/360-370 assembly language has an equivalent pseudo-instruction for each of the pseudo-instructions that we have discussed. The following table lists the pseudo-instructions that are of prime importance. Example 5.4 illustrates how they are used.

PSEUDO-INSTRUCTION	CODE	EXAMPLE	ASSEMBLER ACTION
Origin	ORG	ORG X'100'	PLC ← 100_{16}.
Equate	EQU	X EQU 300	Enter ($X,300_{10}$) in symbol table.
Define storage	DS	Y DS 3F	Move PLC to a full-word boundary if not already at one. Enter (Y,PLC) in symbol table. PLC ← PLC+12.
Define constant	DC	Z DC F'36'	Move PLC to a full-word boundary if not already at one. Enter (Z,PLC) in symbol table. MEMORY[Z; . . .] ← 36_{10}. PLC ← PLC+4.
Using base register	USING	USING *,15	REG[15] is now a base register, and it contains the current value of the PLC.
Drop base register	DROP	DROP 15	REG[15] no longer is to be used as a base register.
End of program	END	END	End of processing.

The origin and equate instructions are identical in function to those for the PDP-11 computer. Example 5.4 shows how the EQU pseudo-instruction is used to identify a register by a symbolic name as the direct-assignment instruction does for the PDP-11.

EXAMPLE 5.4

The following instructions add 10 full words beginning at memory address Z. The left-hand columns show the memory address at which each instruction is assembled and give the operand encodings for the RX instructions. All numbers are given in radix 16.

ADDRESS	R X B D	LABEL	CODE	OPERAND	COMMENTS
			ORG	X'10FE'	PLC ← $10FE_{16}$.
		COUNT	EQU	7	Add (COUNT,7) to symbol table. COUNT will be REG[7].
		INDEX	EQU	8	Similarly, add (INDEX,8) and (SUM,9) to symbol table.
		SUM	EQU	9	
10FE			BALR	15,0	REG[15] ← P+2.
			USING	*,15	Make REG[15] a base register.
1100	7 0 F 054		L	COUNT,TEN	Initialize value of REG[COUNT].
1104	9 0 F 01C		L	SUM,ZERO	Start REG[SUM] at 0.
1108			LR	INDEX,SUM	Start REG[INDEX] at 0.
110A	9 8 F 024	LOOP	A	SUM,Z(INDEX)	Add next item to REG[SUM].
110E	8 0 F 020		A	INDEX,FOUR	Increase REG[INDEX] to next full word.
1112	7 0 F 00A		BCT	COUNT,LOOP	Decrement REG[COUNT] and loop if not 0.
1116	F 0 F 058		B	NEXT	Branch around data.
111C		ZERO	DC	F'0'	
1120		FOUR	DC	F'4'	
1124		Z	DS	10F	
114C		ZNAME	DC	A(Z)	The address of Z.
1150		TEXT	DC	C'TEXT'	Four bytes reserved for characters.
1154		TEN	DC	F'10'	
1158		NEXT	...		

The define storage (DS) and define constant (DC) instructions reserve storage for data. Their PDP-11 counterparts are the . = . +, .WORD, .BYTE, and .ASCII instructions. The DS instruction merely reserves storage; it does not cause any values to be placed in the reserved area. The DC instruction both reserves areas and places initial values in these areas. Both these instructions are used for data of various lengths and representations. The following table gives the assembly language conventions for specifying the length and representation of data:

ASSEMBLY LANGUAGE CODE	DATA TYPE	EXAMPLE	INTERPRETATION
F	Full word, fixed point, radix 10	F'10'	A full word containing 10_{10}
H	Halfword, fixed point, radix 10	H'20'	A halfword containing 20_{10}
D	Double word, fixed point, radix 10	D'35'	A double word containing 35_{10}
X	Digit, radix 16	X'A34E'	Two bytes containing $A34E_{16}$
C	One-byte character	C'ABCD'	Four bytes containing the character codes for "A", "B", "C", and "D"
A	Full-word address	A(Y)	Full word containing address of Y

The description of X-type data requires some clarification. Hexadecimal digits occupy four bits, but the assembler always allocates storage in multiples of eight bits. When an even number of digits is specified, as shown in the table, the storage is allocated with two digits per byte. If the constant contains an odd number of digits, the digit 0_{16} is prefixed to the constant to avoid the problem of allocating a portion of a byte. The various data lengths mentioned in the table are discussed in Section 3.1.

In Chapter 3 we saw that halfwords must begin on halfword boundaries, full words on full-word boundaries, etc. The assembler guarantees that data are placed on proper boundaries. In Example 5.4 the B instruction assembled at 1116_{16} is four bytes long, so that the next item would normally be assembled at $111A_{16}$. However, this is not a full-word boundary, and a full-word datum appears next. For this case the assembler

reserves two extra bytes automatically, so that the full word ZERO is assembled at $111C_{16}$ on a full-word boundary.

The USING instruction notifies the assembler that a register is to be used as a base register and indicates the address to be in it during program execution. In Example 5.4 the BALR instruction places the address of the next instruction in REG[15]. Hence the USING *,15 instruction notifies the assembler that REG[15] will contain the current value of the program location counter, 1100_{16}. The current value of the program location counter is represented by the symbol *. The assembler automatically computes the base and displacement encodings for operands. For example, the instruction L COUNT,TEN must refer to an operand at memory address 1154_{16}. Since REG[15] contains 1100_{16}, the proper effective address can be obtained by using REG[15] as a base register with displacement 54, as indicated in the example. The assembler relieves the programmer of the tedious base-plus-displacement calculations because the programmer may use symbolic names when referring to operands. Of course, the programmer must include instructions to load the base registers properly, but this is much less tedious.

When the contents of a base register are changed, the programmer must notify the assembler. If the register is no longer to be a base register, then the pseudo-instruction DROP must be used. This merely tells the assembler to remove the indicated register from the list of base registers.

The instruction labeled LOOP shows the way to specify indexing of symbolic addresses. Our convention has been to specify index, base, and displacement in the form

displacement(index,base)

When we use symbolic addresses, we no longer specify both base and displacement. For this case we follow the convention

symbolic address(index)

as exemplified by Z(INDEX). Note that Z is the symbolic address that is encoded as REG[15]+24_{16} and that INDEX is REG[8]. Hence the instruction is encoded as if its operand were written X'24'(8,15).

To simplify the task of dealing with constants most assemblers accept operands called *literals*. In assembly language for the System/360-370 computers the literals are data types prefixed by an equal sign, such as =F'3' or =X'10'. When the assembler detects a literal operand, it treats the operand as the memory address of a word that contains the numeric value of the literal. The assembler allocates a memory word for the nu-

meric value, initializes that word to the numeric value, and guarantees that all references to this literal value specify the same address in memory when they are assembled. In general, literals are assigned to locations that follow the last memory location allocated to the program.

EXAMPLE 5.5

The following two sequences of instructions are equivalent:

	LABEL	CODE	OPERAND	LABEL	CODE	OPERAND
1.		L	3,ONE		L	3,='1'
2.	LOOP	A	3,TEN	LOOP	A	3,=F'10'
3.		ST	3,X		ST	3,X
4.		C	3,D41		C	3,=F'41'
5.		BL	LOOP		BL	LOOP
6.		BR	NEXT		BR	NEXT
7.	X	DS	F	X	DS	F
8.	ONE	DC	F'1'	NEXT		
9.	TEN	DC	F'10'			
10.	D41	DC	F'41'			
11.	NEXT					

In this description of System/360-370 assembly language we have barely scratched the surface of the subject. For a full discussion see the System/360-370 reference manual "Assembler Language."

5.3 THE TWO-PASS ASSEMBLER

The assembly process is an interesting example of a principally nonnumeric application of computers. We shall study the assembly algorithm in this section for a better understanding of assembly language. All the elementary processes of an assembler are described in later chapters, so that it should be possible to write an assembler for a small instruction repertoire after completing this text.

The assembly process is shown by the flow charts in Figure 5.1 as a two-pass process. That is, a source program is scanned from start to finish twice during an assembly. Some assemblers are actually one-pass

assemblers. For these assemblers the two phases of processing are performed during a single scan of the program by means of clever programming techniques. Before discussing the details of Figure 5.1, we should note the reason for two-pass processing. Consider the following excerpts from a typical program:

```
        ....
        MOV     X,AC

X:              7
Y:              0

        ...
        MOV     AC,Y
```

The instruction MOV AC,Y can be assembled immediately in a single pass of the text because the memory address of Y has already been determined by the time the assembler encounters the MOV AC,Y instruction. For the MOV X,AC instruction this is not the case, and assembly must be postponed until X is found in the label field. References to symbols to be defined later in a program are sometimes called *forward references.*

In general, a two-pass assembler for PDP-11 assembly language behaves as follows. During the first pass the symbols found in the label field of a program are stored in the symbol table, together with their corresponding addresses. During the second pass the assembler finds the codes for address symbols by searching the symbol table. The codes are combined into words that are output as the assembled program.

Figure 5.1 shows the more important details of the assembly process. In the first pass a symbol appearing in the label field of an instruction is given the value of the location counter and is stored in the symbol table. Next the code field is scanned for pseudo-instructions that might change the program location counter. The = pseudo-instruction leaves the location counter unchanged, and the . =, . = . +, .WORD, .BYTE, and .ASCII pseudo-instructions change the location counter by amounts determined by their operands. Hence during the first pass the code field is inspected, and if the instruction is =, . =, . = . +, .WORD, .BYTE, .ASCII, or .EVEN, special processing takes place. The .END instruction terminates this phase of processing. If the instruction is none of these, the operands are scanned to determine their addressing modes, and the location counter is incremented by 2, 4, or 6 depending on the source and destination addressing modes. This process is repeated for every instruction.

Special processing for pseudo-instructions in the first pass requires

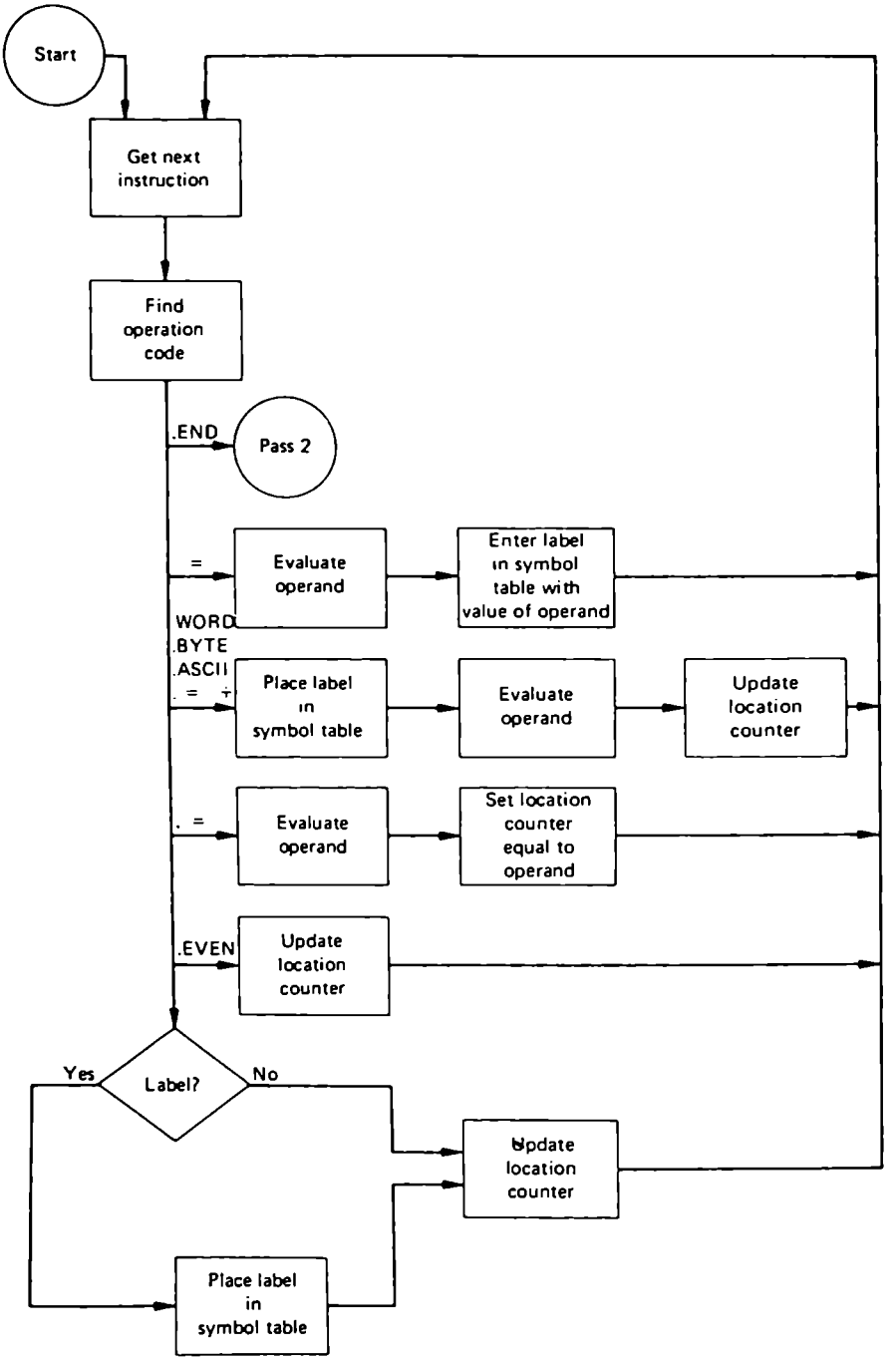


FIGURE 5.1 (a) Flow chart of first pass of assembly

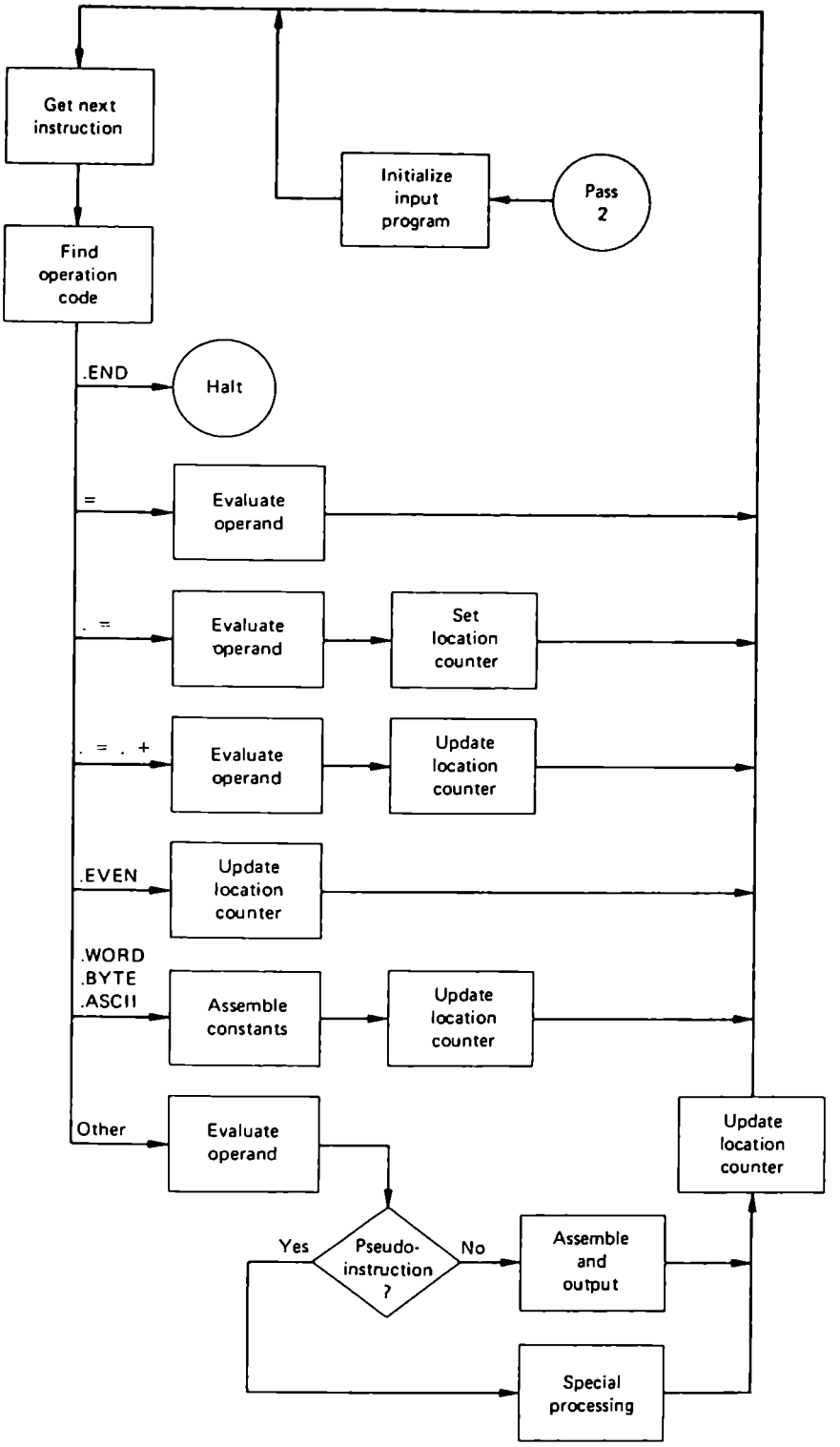


FIGURE 5.1 (*Continued*) (*b*) Flow chart of second pass of assembly

evaluation of the operand field to determine how much the program location counter is to be changed. The increment is computed for each special pseudo-instruction; the program location counter is changed; and the process is repeated for the next instruction.

The second pass is similar to the first pass in general flow of control. In this pass it is not necessary to examine the contents of the label field. In each case the code field is examined to determine whether the instruction is a normal instruction or a pseudo-instruction. If it is a normal instruction, then the code for that instruction is determined by a search through a table that associates each mnemonic code with its binary code. When the operand is evaluated, each symbol in that field is replaced by its value from the symbol table. The operand address is encoded according to its mode and all codes are assembled in the appropriate position in an instruction. In the case of an addressing mode that requires an additional address word, the value of the address word is calculated from the symbols appearing in the operand field and assembled into the address-word position. The PLC is then updated. Zero, one, or two address words are assembled for each instruction depending on the addressing modes specified for the source and destination operands. Finally, all the assembled words for each instruction are output to an external storage medium. Pseudo-instructions encountered in the second pass are processed as special cases.

The assembly process for System/360-370 computers is essentially the same as that described for the PDP-11 computer, except for the following differences:

1. Halfword, full-word, and double-word data must be placed on the proper memory boundaries. The assembler reserves extra space, if necessary, to place data on the correct boundaries.

2. The assembler computes a base-register-plus-displacement encoding of a memory address. During the first pass of the program text, each symbol is stored in the symbol table with its memory address. During the second pass of the text, when it computes the base-plus-displacement encoding for a memory address, the assembler searches a list of base registers active at that point in the program and selects a base register that yields a displacement small enough to be encoded in the 12-bit displacement field. If none exists, the assembler notifies the programmer of an addressing error.

3. The assembler maintains a list of active base registers and their contents by taking note of USING and DROP pseudo-instructions.

5.4 LOADERS AND PROGRAM RELOCATION

As a program is assembled, the assembler usually outputs the encoded program on some output medium such as punched cards, magnetic tape, paper tape, or a magnetic disk. Since the assembler is itself a program that occupies a portion of computer memory, it is usually impractical to place an assembled program in memory with the assembler. Conflicts can arise, for example, if the assembled program must reside in areas that contain portions of the assembler, so that the two cannot be in memory simultaneously. Another problem arises if the size of the assembled program, the assembler, and the symbol table combined is larger than available memory, even though the assembled program alone can reside in memory. Consequently, assembled programs are usually recorded outside computer memory and after assembly must be placed into memory in order to be executed.

For most computers the programs called *loaders* perform the task of loading an assembled program into its proper location in memory. In its most primitive form a loader only reads an assembled program from an external storage medium and places it in the proper area of memory. To facilitate this loading process, assemblers partition the assembled program into blocks of words, and with each block they record information that controls how the block is to be loaded. Usually the information specifies the starting address of the block and the number of words in the block. During loading, the loader examines the control information and places the block in memory at the specified address.

There is a somewhat more versatile type of loader called a *relocating loader*. This type of loader can be used to perform the following tasks in addition to the basic loading operation:

1. A program can be displaced from its assembled address during loading, and the loader will automatically modify the addresses in the program so that the program executes properly in its displaced position.

2. Two or more separately assembled programs can be loaded together in nonconflicting areas of computer memory. Memory addresses in one program that refer to labels in another program can be filled in by the loader so that cross references among programs will be correct.

3. Programs can call on standard algorithms such as sine, cosine, and square root, and the loader will automatically search a library for them.

When relocation is performed by a loader, the result is essentially the same as would be obtained by modifying the origin instruction at the beginning of the program. For example, if the first instruction were . = 100, then the program could be displaced 200 memory cells by changing the instruction to . = 300 and reassembling. The same result could be obtained by using a relocating loader to load the version of the program assembled at 100 with a displacement of 200. The advantage of using a relocating loader is that the program need not be reassembled in order to relocate it, and relocation is considerably simpler than assembly.

To see what is involved in relocation consider the assembly language programs below. The first is in PDP-11 assembly language, and the second is in System/360-370 assembly language.

If the first program were displaced by 200 memory cells, what changes would there be in the encoded instructions? The address-modifying words for the MOV instructions do not change, since the positions of X and Y relative to the instructions remain constant. The contents of ZNAME clearly have to be changed, since ZNAME contains the address of Z. The general rule for relocation in PDP-11 programs is to add the amount of displacement to every word that contains the absolute address of another word in memory. Note that the displacement must be added, not to every word in memory, but only to those words that contain addresses of words. The words labeled X, Y, and Z contain constants, and we must not alter these constants when the program is relocated. Address words for relative addresses do not have to be changed when a program is relocated, so that it is possible to construct programs that are relocatable without change by using this address mode exclusively for references to operands in memory. Other address modes for operands might require alterations to address words.

```
LABEL     CODE    OPERAND       COMMENTS
-------------------------------------------------------
          =       100
          MOV     X,AC          ; Load X into the accumulator.
          MOV     AC,Y          ; Store AC into Y.
          MOV     AC,@ZNAME     ; Store AC into location pointed to
                                ; by ZNAME.
          HALT                  ; Stop.
X:        5
Y:        9.
Z:        0
ZNAME:    Z                     ; The address of Z.
          .END
```

```
LABEL    CODE    OPERAND     COMMENTS
         ORG     100
TEMP     EQU     2
         BALR    15,0        REG[15] ← P+2.
         USING   *,15        Establish REG[15] as base register.
         L       TEMP,X      REG[TEMP] ← X.
         ST      TEMP,Y      Y ← REG[TEMP].
         . . .
X        DC      F'5'
Y        DC      F'0'
XNAME    DC      A(X)        The address of X.
         END
```

Now we ask the same question about the second program. Do we have to change the encoding of memory address X in the L instruction? The answer to this question is negative. The address in the L instruction uses REG[15] as a base register, and the value in that register will always be the address of the L instruction, no matter where the program is relocated in memory. Hence the effective address for the L instruction will always be the correct memory address. Note that the address assembled in the cell labeled XNAME is affected by relocation. XNAME contains the true memory address of X, and not the address in base-register-plus-displacement form. The general rule for relocation in System/360-370 programs is to adjust all addresses appearing in DC pseudo-instructions by the amount of the relocation.

We see from the examples that relocation involves adding a constant to some, but not all, words in an assembled program. Unfortunately, the loader cannot examine a word in an assembled program and decide whether it contains an instruction, a constant, or an address. If no other information were available to it, the loader could not determine which words should be modified and which should not be when a program is relocated. Therefore, the assembler inserts in the assembled program additional information that specifies which words contain addresses that must be modified. The loader uses this information when it performs relocation.

The second capability of relocating loaders concerns the loading of two separately assembled programs so that they can refer to each other. For example, if one program has the instruction MOV X,AC and another has the instruction X: 0, the assembler cannot determine the memory address for the MOV instruction unless the second program is assembled with the first. By means of the relocating loader we can place such ex-

ternal references in programs and still assemble them separately. In essence we defer the assembly of external references until the programs are loaded.

We use pseudo-instructions as follows for external references: the first is for the PDP-11 computer, and the next two are for the System/360-370.

COMPUTER	PSEUDO-INSTRUCTION	CODE	EXAMPLE	ACTION
PDP-11	Global	.GLOBL	.GLOBL ABC	Mark ABC as a symbol either defined in another program or used in another program.
360-370	External	EXTRN	EXTRN ABC	Mark ABC as a symbol defined in another program.
360-370	Entry	ENTRY	ENTRY ABC	Mark ABC as a symbol defined in this program that is used by another program.

Programs that refer to external memory addresses must use the .GLOBL or EXTRN instructions to notify the assembler that the addresses are external to the program. The assembler places this information in the assembled program together with other information that permits the relocating loader to modify the memory addresses during loading. The information included in the assembled program is each external symbol and a list of the addresses of all instructions that refer to that symbol.

Similarly, if a program contains locations referenced by other programs, the program must notify the assembler by using the .GLOBL or ENTRY pseudo-instructions. An assembler processes these instructions by placing the symbol and its address into the assembled program. Examples 5.6 and 5.7 show how the .GLOBL instruction is used in a program for the PDP-11. Examples 5.8 and 5.9 depict the use of the EXTRN and ENTRY instructions for the System/360-370.

When a set of PDP-11 programs with external references is loaded, the loader places all .GLOBL references in a symbol table, together with the addresses at which the .GLOBL symbols are loaded. When all programs have been processed, the loader examines the list of .GLOBL symbols. For each reference to a .GLOBL symbol the loader finds the address in the symbol table and replaces the reference by this address. The

processing for the .GLOBL is essentially the same as the processing that takes place during assembly. The loader uses the list of .GLOBL symbols only to construct addresses for external references. When loading is completed, the lists need not be retained in memory with the program. Similar processing occurs for System/360-370 computers with ENTRY used to build the symbol table and EXTRN used to find references in it.

EXAMPLE 5.6

The following PDP-11 program is assembled separately from the program in Example 5.7. Symbols XYZ and ABC refer to locations that are not in the following PDP-11 program.

```
LABEL    CODE      OPERAND
         .GLOBL    XYZ,ABC
         ...
         ADD       XYZ,%2
         ...
         MOV       ABC,AC
         ...
```

EXAMPLE 5.7

The following PDP-11 program is to be loaded together with the program of Example 5.6:

```
LABEL    CODE      OPERAND
          GLOBL    XYZ,ABC
         ...
XYZ:     MOV       AC,TEMP
         ...
ABC:               26
```

EXAMPLE 5.8

Symbols XYZ and ABC refer to locations that are not in the following System/360-370 program:

```
LABEL    CODE      OPERAND
         EXTRN     XYZ,ABC
         ...
         A         7,XYZ
         ...
         L         3,ABC
```

EXAMPLE 5.9

The following System/360-370 program is to be loaded along with the program of Example 5.8:

```
LABEL    CODE      OPERAND
         ENTRY     XYZ,ABC
         ...
XYZ      ST        TEMP,4
         ...
ABC      DC        F'26'
         ...
```

The third capability of relocating loaders is library searching and loading. A program library is a collection of frequently used programs that is stored externally to the computer on a magnetic disk, tape, or some equivalent storage medium. The programs in the library are stored in their assembled forms, as relocatable programs, so that it is not necessary to reassemble a library program each time it is used. Since references to library programs are really no different from any other external references, library processing is a natural extension of external-reference processing. The usual algorithm for relocating loaders is to replace as many external references as possible with addresses from the loader symbol table. If external references are left over, the loader searches a library for programs that contain the missing references. Thus a programmer

can call for an algorithm for computing the sine by using .GLOBL SINE or EXTRN SINE provided that the sine program has been assembled for the library with the instruction .GLOBL SINE or ENTRY SINE.

We have discussed relocating loaders from a rather general point of view in this section. There are many loaders that differ somewhat from the description here. Notable exceptions are the loaders for the PDP-11 and the System/360-370 computers. For these computers relocation, external-reference processing, and library searching are performed by a program called a *link editor*. The link editor uses the output from the assembler, often called an *object module*, and produces a version of the program that essentially needs only loading, called a *load module*. The loader uses the load module and can perform only loading and simple relocation. In effect, the processes we have attributed here to a single program, the relocating loader, are split between the link editor and the loader for the PDP-11 and System/360-370.

5.5 MACRO-INSTRUCTIONS

There are many situations in which the programmer must tediously reproduce sequences of assembly language instructions at several different places in a program. For example, he may wish to increment a positive number in memory by 3. One way of doing this for the PDP-11 computer is to use the INC instruction repeatedly as follows:

```
...
INC   X    Increment X by 1.
INC   X
INC   X
```

Ideally the programmer wants to tell the assembler how to construct the sequence of instructions without specifying the instruction separately each time. In this example he ought to be able to tell the assembler to repeat the instruction INC X three times. Many assemblers have facilities for generating sequences of instructions under control of the programmer. Instructions that cause other instructions to be generated are commonly known as *macro-instructions*, or sometimes *macros*.

Both the PDP-11 (the Macro-11 assembler) and System/360-370 assembly languages permit a programmer to define a macro-instruction by giving a sequence of commands for that instruction. Then whenever the macro-instruction is used, the assembler replaces the macro-instruction by the sequence of instructions used in its definition. For example, in the PDP-11 a left cyclic shift, excluding the carry bit, can be performed by the following sequence of instructions. The single-operand add

with carry instruction (code ADC) adds the condition code register C to the operand.

```
ASL    X    ; Logical shift X left.
ADC    X    ; Set the least significant bit of X to the value of the condition
            ; code register C.
```

If we wish to perform a left cyclic shift at several places in a program, it becomes convenient to define a left cyclic shift macro-instruction. The following example shows how we define a macro-instruction LSHIFT for left cyclic shift:

```
.MACRO   LSHIFT Z   ; Beginning of macro-definition that includes the
                    ; macro-name LSHIFT and its one operand named
                    ; Z.
         ASL    Z   ; Use Z as the operand.
         ADC    Z
         .ENDM       ; End of macro-definition.
```

After defining LSHIFT as shown, the programmer can create the sequence of instructions for computing the left cyclic shifts by using the following instruction:

```
LSHIFT Y
```

When the assembler reaches the LSHIFT instruction, it replaces it by the definition of LSHIFT. In the original definition of LSHIFT the operand is named Z and is identified as an operand of the macro by its appearance in the list of operands. Every reference to Z is changed to Y, because Y is specified as the operand for this use of the macro. Thus the LSHIFT Y instruction produces the instructions

```
ASL Y and ADC Y
```

The general format of a macro-instruction is as follows:

```
.MACRO   NAME   OP1,OP2, . . . ,OPN
         INSTR1
         INSTR2
         INSTR3
         . . .
         INSTRm
         .ENDM
```

The definition begins with .MACRO and ends with .ENDM. The first line also specifies the name by which the macro will be invoked (NAME) and dummy names for the operands. The macro may be invoked by a call such as

```
NAME R1,R2,R3,R4
```

This call causes all the instructions in the definition to be inserted into the program at the calling point, with OP1 replaced by R1, OP2 replaced by R2, etc.

The macro facility for the System/360-370 assembly language is very similar to that for the PDP-11. The general format of a System/360-370 macro-instruction is as follows:

```
          MACRO
&LABEL    NAME      &OP1,&OP2,. . .,&OPn
          INSTR1
          INSTR2
          INSTR3
          . . .
          INSTRm
          MEND
```

The definition begins with MACRO and ends with MEND. The first line after MACRO specifies the name by which the macro is invoked (NAME), dummy names for the operands, and an optional dummy label name. The macro is invoked by a call such as

```
LOOP NAME R1,R2,R3,R4
```

This call causes all the instructions in the definition to be inserted into the program at the calling point, with &OP1 replaced by R1, &OP2 replaced by R2, etc., and &LABEL replaced by LOOP. Presumably, &LABEL appears in the label field of exactly one instruction in the macro-definition.

EXAMPLE 5.10

The following macro-definition tests the full-word operand &X to determine whether it is odd or even, leaving the result of the test in the condition code. A branch-on-zero instruction (code BZ) can be

used after the EVEN macro-instruction, with the branch taken if the operand is even and no branch taken if the operand is odd.

LABEL	CODE	OPERAND	COMMENTS
	MACRO		
&START	EVEN	&X,&TEMP	EVEN is name of macro.
&START	L	&TEMP,&X	REG[&TEMP] ← MEMORY[&X; . . .].
	N	&TEMP,=F'1'	REG[&TEMP] ← REG[&TEMP] AND 1.
	LTR	&TEMP,&TEMP	Test REG[&TEMP] and set condition code.
	MEND		

If the macro is invoked by the instruction

```
TESTZ EVEN Z,5
```

then the following instructions are generated by the assembler:

LABEL	CODE	OPERAND
TESTZ	L	5,Z
	N	5,=F'1'
	LTR	5,5

The label TESTZ can be the target of a branch instruction elsewhere in the program. In this example the dummy label feature is used to distinguish this invocation of EVEN from all others.

Macro-definitions may contain macro-instructions as well as ordinary instructions or pseudo-instructions. For example, a programmer can define a macro-instruction that uses the macro EVEN, defined in Example 5.10. Macro-instructions embedded in other macro-definitions are known as *nested macros.*

REFERENCES

The PDP-11 and System/360-370 assembly languages contain several facilities in their macro-language that are not treated here. One of the most useful of these is the set of instructions for conditional assembly that

can be used to generate instructions selectively. These instructions can omit the generation of particular instructions a variable number of times, with the generation controlled by the value of the operands of the macro at the point it is invoked. For a full discussion consult the PDP-11 Macro Assembler Form DEC-11-OMACA-BD or the System/360-370 reference manual, "Assembler Language," form C28-6514, respectively. A tutorial discussion of the System/360-370 macro-language appears in Kent (1969).

EXERCISES

5.1 If line 8 of Example 5.1 were deleted, what changes in the address portions of instructions would have to be made for the address references to remain correct? Assume that the storage addresses Y and Z do not change. The use of symbolic names frees the programmer from the task of changing instruction encoding when he adds or deletes instructions.

5.2 Explain why the pseudo-instructions =, . =, and . = . + do not allow forward references in their operand fields.

5.3 A programmer attempted to execute a program containing the instruction sequence

```
LABEL   CODE    OPERAND
        MOV     X,AC
X:              10.
        MOV     AC,Y
        HALT
```

What is incorrect in this sequence if the intent is to store the number 10 in MEMORY[...;Y]?

5.4 Determine the instruction encoding for the program in Example 5.5 under the assumption that the instruction in line 1 is immediately preceded by the pseudo-instruction ORG X'4000'.

5.5 Here is a sequence from a PDP-11 program that assembled properly:

```
     LABEL     CODE     OPERAND
-------------------------------
                ...
1.   AC        =        %0
2.             MOV      X,AC
                ...
3.   X:                  0
```

When instructions 1 and 2 were interchanged and assembly attempted, a phase error was encountered. The definition of a phase error is: "A label's definition or value varies from pass one to pass two of the assembler." Explain why the phase error occurred.

SIX

DATA STRUCTURES

The processing of data structures is of central importance in computer programming. Data structures are collections of data that are related in some manner. Vectors and matrices are examples of data structures in which the data have a positional relationship. A library card catalog is a data structure of a more complex type. A book listed in it may be cross-referenced to other books on the basis of a related subject, a common author, or both. Moreover, a single book may be included in several sets or groupings of books each of which has a different basis of relationship.

Although there are many structures that satisfy our vague definition, we shall confine this discussion to a few data structures of general importance. For computer-programming purposes nearly every data structure of interest can be manipulated by casting it into one of the data structures we describe here.

The study of data structures is a fundamental part of the study of algorithms and programs. Just as numbers can be represented in several different systems, so a set of related data can be fitted into many different data structures. When a programmer creates an algorithm to solve a problem, he also creates a data structure that is manipulated by the algorithm. His choice of data structure can greatly affect the amount of storage required for processing, as well as the processing speed. It is common to find that memory requirements may vary by a factor of 2 to 10 for different structural representations of the same data. Moreover, because the data structure affects the processing of the data, computa-

tional speeds may vary by even larger factors. If N is some natural parameter of a problem, the computation time for one data structure may be proportional to N^2, but for another it may be only $N \log_2 N$. Clearly, as N becomes large, the differences in processing speed become extremely large. In some instances it is possible to exhibit computation times that vary from as high as $N!$ or 2^N to times that grow only algebraically in N, say N^5. The improvement in processing speed of efficient algorithms is often largely due to the use of appropriate data structures.

In this chapter we shall initiate the study of data structures by investigating some fundamental ones. Later chapters illustrate how these data structures can be used in practice. There are several features of the PDP-11 and IBM/360-370 that simplify the manipulation of various data structures. These features are described with the appropriate data structure.

6.1 ONE-DIMENSIONAL ARRAYS

The simplest data structure is the *one-dimensional array*. An array may be viewed as a list of items such that each item is identified uniquely by its index in the list. For example, let X be the label for an array with 100 elements whose indices range from 0 to 99. Then X[51] is an item near the middle of the array, X[0] is the first item, and X[99] is the last.

To take advantage of the characteristics of addresses in computers, it is customary to store array elements in sequence in memory so that the address of X[1] can be calculated by adding a constant to the address of X[0]. The following PDP-11 assembly code instruction creates an array with 100_{10} words and associates the name X with the first word, X[0], of the array.

```
X: . = . + 200.
```

Since the array elements are words, the address of X[1] is derived by adding 2 to the address of X[0].

The index addressing mode makes array addressing particularly easy for the PDP-11. Recall that in the index addressing mode the effective address is the sum of an address word and an index register.

Indexing is indicated symbolically by X(R), where R is a register and X is the value of the address word. The effect of the instruction

```
MOV %3,X(%1)
```

is:

```
comment fetch the address word;
MAR ← P;
```

```
P ← P+2;
MDR ← MEMORY[...;MAR];
comment compute effective address;
MAR ← MDR+REG[1];
comment store REG[3] at the index address;
MEMORY[...;MAR] ← REG[3];
```

Scanning an array in ascending order with the PDP-11 computer instruction set can be done conveniently by means of index addressing. Example 6.1 illustrates one way of scanning an array.

EXAMPLE 6.1

The program finds the maximum element of an array X, leaving its index in REG[1] and its value in REG[0].

```
    LABEL     CODE    OPERAND        COMMENTS
1.  MAX       =       %0             ; REG[MAX] will hold value
                                     ; of maximum.
2.  PLACE     =       %1             ; REG[PLACE] will hold index
                                     ; of maximum element.
3.  INDEX     =       %2             ; The current index.
4.  TEMP      =       %3             ; Temporary register.
5.  LNGTH:            100.           ; Length of X is placed here.
6.  X: .      =       . + 200.       ; Reserve storage for X, 200
                                     ; bytes.
            ...     ...                ...
7.          CLR     PLACE          ; Assume maximum in X[0].
8.          MOV     X,MAX          ; Store value of X[0] in MAX.
9.          CLR     INDEX
10.         INC     INDEX          ; Start loop at X[1].
11. ROUND:  MOV     INDEX,TEMP
12.         ASL     TEMP           ; Multiply the index by 2.
13.         CMP     X(TEMP),MAX    ; Is X[TEMP] ≤ MAX?
14.         BLE     TEST           ; If so, branch; otherwise
                                   ; continue.
15.         MOV     X(TEMP),MAX    ; Update maximum value.
16.         MOV     INDEX,PLACE    ; Save index.
```

```
    LABEL     CODE    OPERAND          COMMENTS
17. TEST:     INC     INDEX            ; Increment current index.
18.           CMP     INDEX,LNGTH      ; Is current index < LNGTH?
19.           BLT     ROUND            ; If so, branch.
              ...
```

In this example the loop control is the index and the limit of the loop is the maximum allowable index. The index is in REG[INDEX] at line 11. To compute the address of X[INDEX], we have to multiply the index by 2, since each element of the full-word array X is two bytes long. When a new maximum is found, MAX and PLACE are updated.

Note carefully how the limit is tested in lines 18 and 19. The jump out of the loop is taken when INDEX $\geq$ LNGTH. The jump out of the loop could be taken if INDEX = LNGTH rather than INDEX $\geq$ LNGTH. If LNGTH is greater than zero, it makes no difference which test is used. However, suppose that LNGTH is zero. Then the value of the index when the first comparison occurs is 1, so that an equality test would cause 65,535 iterations to occur. An inequality test would result in an immediate termination.

The lesson is that the programmer should consider program behavior under abnormal conditions. The equality test is sensitive to bad data because it requires exact equality. It should be used only if a check is inserted at the beginning of the program to be sure that the data will not cause a loop to occur.

Example 6.2 is the same algorithm as Example 6.1, but written for System/360-370.

EXAMPLE 6.2

```
    LABEL     CODE    OPERAND          COMMENTS
1.  MAX       EQU     2                REG[MAX] will hold value of
                                       maximum.
2.  PLACE     EQU     3                REG[PLACE] will hold index of
                                       maximum element.
3.  TEMP      EQU     4                A temporary register.
4.  INDEX     EQU     5                The current index.
5.  XNAME     DC      A(X)             The memory address of X.
6.  LNGTH     DC      F'100'           The length of X.
```

```
     LABEL    CODE    OPERAND          COMMENTS
7.   X        DS      100F'0'          Reserve 100 full words for X.
                 . . .
8.            L       MAX,X(0)         Initialize MAX and PLACE as if
                                       X(0) contained maximum.
9.            L       PLACE,=F'0'
10.           L       INDEX,=F'1'      Start loop at X[1].
11.  ROUND    LR      TEMP,INDEX       Multiply the index by 4. A full
                                       word consists of 4 bytes.
12.           SLA     TEMP,2
13.           C       MAX,X(TEMP)      Is MAX < X[INDEX]?
14.           BNL     TEST             If not, branch; otherwise con-
                                       tinue.
15.           L       MAX,X(TEMP)      Update maximum.
16.           LR      PLACE,INDEX
17.  TEST     A       INDEX,=F'1'      Add 1 to index.
18.           C       INDEX,LNGTH      Is INDEX < LNGTH?
19.           BL      ROUND            If so, branch.
                 . . .
```

Examples 6.1 and 6.2 illustrate the great similarities of the index addressing mode in the PDP-11 and the base-displacement-index addressing in the System/360-370. The addressing is virtually the same in both cases. For an array computation, the address word of the PDP-11 holds the corresponding information held by a base register in the System/360-370. The index registers serve the same function in both computers. The PDP-11 has no displacement field.

Another way of implementing Example 6.1 in PDP-11 assembly language makes use of the autoincrement addressing mode and simplifies the programming somewhat. Recall that in the autoincrement addressing mode, a register is designated whose contents point to an operand. After the operand has been accessed, the contents of the register are incremented by 2 so that they point to the next operand.

Autoincrement mode is indicated symbolically by (R)+. The effect of the instruction ADD (%1)+,%2 is:

```
comment REG[1] has the source address. Fetch the source operand;
MAR ← REG[1];
MDR ← MEMORY[. . . ;MAR];
comment increase REG[1];
REG[1] ← REG[1]+2;
```

comment add source operand to REG[2];
REG[2] ← REG[2]+MDR;

The autoincrement mode is very useful when the elements of a one-dimensional array have to be processed sequentially. Here Example 6.1 is repeated below using autoincrement mode.

Examples 6.1 to 6.3 illustrate how to access elements of an array

EXAMPLE 6.3

The program finds the maximum element of an array X, leaving its index in REG[1] and its value in REG[0].

	LABEL	CODE	OPERAND	COMMENTS
1.	MAX	=	%0	; The maximum value.
2.	PLACE	=	%1	; The index of the maximum ; value.
3.	INDEX	=	%2	; The index variable for the ; loop.
4.	XI	=	%3	; The address of X[INDEX].
5.	LNGTH:		100.	; The loop limit.
6.	X:.	=	. + 200.	; Save 200 bytes for X.
7.	X1:		X + 2	; Contains address of X[1].
	...	...	...	...
8.		CLR	PLACE	
9.		MOV	X,MAX	; Initialize MAX as the first ; element of the array.
10.		MOV	X1,XI	; REG[XI] now has address of ; X[1].
11.		CLR	INDEX	
12.		INC	INDEX	; Set INDEX to second ele- ; ment.
13.	ROUND:	CMP	(XI)+,MAX	; The item designated by IN- ; DEX, X[INDEX], is fetched ; and compared to MAX. The ; pointer XI is increased by a ; value of 2.
14.		BLE	TEST	; If X[INDEX] $\leq$ MAX then ; branch; otherwise continue.

```
     LABEL     CODE    OPERAND          COMMENTS
15.            MOV     -2(XI),MAX       ; Update MAX. XI has been in-
                                        ; cremented two bytes past
                                        ; last item.
16.            MOV     INDEX,PLACE      ; Update PLACE.
17.  TEST:     INC     INDEX            ; Increment current index.
18.            CMP     INDEX,LNGTH      ; Is current index < LNGTH?
                                        ; If so, branch.
19.            BLT     ROUND
               ...
```

when the array has a lower bound of 0. It is common practice to give the first element of an array an index other than 0. For example, consider a program with an array named POPULATION that holds census data for the years 1950 to 1964. Since the data cover only 15 years, we want the array to be 15 words long. It may be desirable to index the elements of the array so that POPULATION[1950] holds the size of the population in 1950. However, we obviously do not want to leave 1950 memory cells blank in order to have the desired indexing. Hence we create the array so that POPULATION[1950] is the first element of the array or, equivalently, 1950 is the lower bound for POPULATION.

The FORTRAN programming language defines all arrays to have a lower bound of 1. Thus the FORTRAN statement

```
DIMENSION X(100)
```

creates an array named X with 100 elements such that the first has index 1 and the last has index 100. ALGOL permits the programmer to specify both the upper and lower bounds of the arrays he defines.

In general, then, every one-dimensional array has two integers associated with it: a lower bound and an upper bound. These are indices of the first and last elements of the array, respectively. The length of the array is

$$\text{length} = \text{upper bound} - \text{lower bound} + 1$$

It is frequently worthwhile to test indices to determine if they are in range before using them. Such tests expose a very common source of programming errors.

Example 6.4 shows two sequences of instructions for checking subscripts and for computing the address of an array element. The general rules to follow are:

1. Take error exit if index does not satisfy
   lower $\leq$ index $\leq$ upper bound
2. The indexed address is obtained by computing
   base + index − lower bound

EXAMPLE 6.4

The first sequence is for the PDP-11 computer and the second is for System/360-370. In the first sequence the index is checked, then 2 × (index − lower bound) is used as the address of the indexed array element.

```
    LABEL     CODE    OPERAND      COMMENTS
1.  INDEX:            0            ; Index is assumed to be here.
2.  LOWER:            1            ; Lower bound is index of first ele-
                                   ; ment.
3.  UPPER:            100.         ; Upper bound is index of last ele-
                                   ; ment.
4.  BASE:             ARRAY        ; Address of first element.
5.  ARRAY:.   =       .+200.       ; Array storage reserved here.
6.  I         =       %0           ; Define I as REG[0].
    ...       ...     ...            ...
7.            MOV     INDEX,I      ; Fetch value of index from mem-
                                   ; ory.
8.            CMP     I,UPPER      ; Is INDEX > UPPER?
9.            BGT     ERROR        ; If so, branch out.
10.           SUB     LOWER,I      ; Compute I — LOWER and set con-
                                   ; dition codes.
11.           BLT     ERROR        ; If INDEX < LOWER then branch
                                   ; out.
12.           ASL     I            ; Multiply INDEX by 2 to convert to
                                   ; word.
13.           ADD     BASE,I       ; REG[0] now contains the address
                                   ; of element ARRAY[INDEX].
```

The following sequence, for System/360-370, checks the index and then places 4 × (index − lower bound) in REG[1] so that REG[1] can be used as an index register in another instruction.

	LABEL	CODE	OPERAND	COMMENTS
1.	INDEX	DC	F'0'	
2.	LOWER	DC	F'1'	Lower bound is index of first element.
3.	UPPER	DC	F'100'	Upper bound is index of last element.
4.	BASE	DC	A(ARRAY)	Address of first element.
5.	ARRAY	DS	100F	Reserve storage for array.
6.	I	EQU	2	Define I as REG[2].
		...	...	...
7.		L	I,INDEX	Fetch value of index from memory.
8.		C	I,UPPER	Is INDEX < UPPER?
9.		BH	ERROR	If so, branch out.
10.		S	I,LOWER	Compute I — LOWER.
11.		LTR	I,I	Is INDEX < LOWER? (This instruction is redundant since previous instruction also sets condition code.)
12.		BL	ERROR	If so, branch out.
13.		SLA	I,2	Multiply by 4 by shifting twice.

In some cases the lower bound is known to be 0. For these cases it is not necessary to subtract a lower bound from the index. For example, in the System/360-370 sequence in Example 6.4 line 10 can be deleted.

6.2 MULTIDIMENSIONAL ARRAYS

There are several ways to generalize the addressing of one-dimensional arrays to two or more dimensions. We shall consider two alternatives here.

6.2.1 The Address Polynomial Method of Indexing

Suppose that X is the name of an $M \times N$ matrix, and suppose that we consider the general problem of addressing X[I,J], the element of X in row I, column J. If we store X in a particular fashion, then we can compute the address of X[I,J] from the values of I and J by a multiplication and addition. A suitable storage format is to store X[1,1] at the base address of X, and at successive locations we store

X[1,2],X[1,3], . . .,X[1,N], X[2,1], . . .,X[2,N], . . .,X[M,N]

Then the address of X[I,J] is given by

$$\text{address} = \text{base} + [(I - 1)N + J - 1]$$

The bracketed part of the expression is often called the *address polynomial.* In this example the elements of X are said to be stored in *row-major form*, because X is stored by rows in memory. It is equally acceptable to store X by columns (or *column-major form*) and to calculate the address of X[I,J] by use of the polynomial

$$\text{address} = \text{base} + [(J - 1)M + I - 1]$$

Addressing of this type is usually done in the machine language programs compiled by FORTRAN compilers. (In fact, the ANSI FORTRAN standard requires that arrays be stored in column-major form.) If we permit arbitrary lower bounds and also require checking of indices before indexing an address, then the following general rules apply:

1. The index in each dimension must satisfy the relation
   lower bound $\leq$ index $\leq$ upper bound
2. For each dimension calculate an effective index by subtracting the lower bound for that dimension from the value of the index.
3. For each dimension compute a length from the formula
   length = upper bound − lower bound + 1
4. The address polynomial is computed for a k-dimensional array from the formula

   address polynomial
   $$= ((\cdots(E_1)L_2 + E_2)L_3 + \cdots + E_{k-1})L_k + E_k$$

   where the E_i are the effective indices and the L_i are the lengths.

6.2.2 Addressing by Indirection

Another form of addressing avoids the use of multiplications in the calculation of addresses within multidimensional arrays. It is best illustrated by an example in the two-dimensional case. Let X be an $M \times N$ array as before, and assume that X is stored in row-major form. To index into X we use an array named XROW of length M. The Ith entry of XROW contains the address of X[I,1], the first entry of the Ith row of X. Then the address of X[I,J] is given by

$$\text{address} = \text{MEMORY}[\ldots; (\text{base of XROW}) + I] + J$$

The following example illustrates addressing into an array with indices I and J with $0 \leq I \leq 4$ and $0 \leq J \leq 2$.

Note that, in line 17 of the PDP-11 program, the deferred register mode is used to fetch the operand because REG[TEMP] holds the *address* of the operand, and not the operand.

The programs in Example 6.5 can be expanded to include checks to ensure that the indices are in range, and they can be generalized to the case in which lower bounds are nonzero.

EXAMPLE 6.5

The first of the two sequences is for the PDP-11 computer, and the second is for System/360-370. In the code sequences, note the multiplication by 2 and 4, for the PDP-11 and System/360-370, respectively, to compute addresses from indices.

```
     LABEL   CODE   OPERAND            COMMENTS
1.   XROW:          X
2.                  X + 6.             ; Address of third word be-
                                       ; yond X.
3.                  X + 12.            ; Address of sixth word be-
                                       ; yond X, etc.
4.                  X + 18.
5.                  X + 24.
6.   X:     =       + 30.
7.   TEMP   =       %0                 ; A temporary register.
8.   JAY    =       %1                 ; REG[JAY] will hold value
                                       ; of J.
9.   I:             0
10.  J:             0
                .  .  .
11.             MOV    I,TEMP          ; Fetch I to REG[TEMP].
12.             ASL    TEMP            ; Multiply by 2.
13.             MOV    XROW(TEMP),TEMP ; Fetch address of row I.
14.             MOV    J,JAY
15.             ASL    JAY             ; Make J into an index.
16.             ADD    JAY,TEMP        ; Form address for X[I,J].
17.             MOV    @TEMP,TEMP      ; Fetch X[I,J].
```

The following is the same program written for System/360-370.

	LABEL	CODE	OPERAND	COMMENTS
1.	XROW	DC	A(X)	
2.		DC	A(X+12)	Address of third full word beyond X.
3.		DC	A(X+24)	
4.		DC	A(X+36)	
5.		DC	A(X+48)	
6.	X	DS	15F	
7.	TEMP	EQU	3	A temporary register.
8.	JAY	EQU	4	REG[JAY] will hold value of J.
9.	I	DC	F'0'	
10.	J	DC	F'0'	
		. . .		
11.		L	TEMP,I	Fetch I to REG[TEMP].
12.		SLA	TEMP,2	Multiply by 4.
13.		L	TEMP,XROW(TEMP)	Fetch address of row I.
14.		L	JAY,J	
15.		SLA	JAY,2	Make J into an index.
16.		L	TEMP,0(JAY,TEMP)	Fetch X[I,J].

6.3 QUEUES

Queues are one-dimensional arrays used for temporary storage of data. Items are entered and removed one at a time such that the first item placed into the queue is the first item to be removed. Often queues are called *first-in-first-out* storage structures. Two variables are associated with queues: the variable HEAD identifies the next item to be retrieved, and the variable TAIL identifies the place to put the next item to be added to the queue. The processes are called by many different names, such as QUEUE and DEQUEUE. We shall use the names PUT and GET.

Let us assume that the queue is implemented in an infinite array named QUEUE and that GET takes one item from the head of the queue and places it in the register AC, while PUT takes one item from register AC and places it at the tail of the queue. The ALGOL-like descriptions of PUT and GET may be written as follows:

```
PUT:  TAIL ← TAIL+1;
      QUEUE[TAIL] ← AC;
```

```
GET:   if HEAD < TAIL then
           begin
               HEAD ← HEAD+1;
               AC ← QUEUE[HEAD];
           end
       else
           go to QUEUE EMPTY;
           . . .
```

HEAD and TAIL are initialized to the same value. The routine GET retrieves an item from QUEUE, provided that at least one more item has been stored in the queue than has been retrieved.

The routines are actually more complex than we have indicated. The array QUEUE is finite, so that the permissible values of HEAD and TAIL must be limited. Queueing is generally implemented by indexing from 0 to some maximum value designated MAX; and when either HEAD or TAIL surpasses the value of MAX, it is reset to 0. We have to be very careful that the indices HEAD and TAIL never pass each other. In the rough description above, the condition that HEAD increases beyond the value of TAIL indicates that the queue has been emptied and a special test ensures that this can never occur. If, however, TAIL increases beyond the value of HEAD (as it might, since HEAD can be reset to 0), then MAX+1 more PUTs have occurred than GETs, so the queue is full. The following examples illustrate queue processing in its full generality:

```
PUT:   TAIL ← TAIL+1;
       if TAIL > MAX then TAIL ← 0;
       if TAIL = HEAD then go to QUEUE FULL;
       QUEUE[TAIL] ← AC;
GET:   if HEAD = TAIL then go to QUEUE EMPTY;
       HEAD ← HEAD+1;
       if HEAD > MAX then HEAD ← 0;
       AC ← QUEUE[HEAD];
       . . .
```

Note that the comparisons of HEAD and TAIL must be equality comparisons. The circular characteristics of the queue can produce situations in which HEAD > TAIL, for example, and therefore prevent a test of the form HEAD ≥ TAIL for queue emptiness. Example 6.6 depicts the implementation of PUT and GET.

EXAMPLE 6.6

The first sequence is an implementation using the index addressing mode for the PDP-11. The second sequence is for the System/360-

370. TAIL is always the index of the next empty place to add an element to the queue, and HEAD is the index of the next element to be removed. Items in the queue are assumed to be full words, and MAX is measured in words. QUEUE is the starting address of the queue area in memory.

```
    LABEL    CODE    OPERAND          COMMENTS
1.  AC       =       %0               ; The accumulator is
                                      ; REG[0].
2.  TAIL     =       %1               ; The TAIL index is in
                                      ; REG[1].
3.  HEAD     =       %2               ; The HEAD index is in
                                      ; REG[2].
4.  TEMP     =       %3               ; A temporary register.
5.           CLR     TAIL             ; Initialize queue.
6.           CLR     HEAD
    ...      ...     ...                ...
7.  PUT:     INC     TAIL             ; Increment TAIL index.
8.           CMP     TAIL,MAX         ; Is TAIL > MAX?
9.           BLE     NEXT             ; If not, continue.
10.          CLR     TAIL             ; Otherwise clear TAIL to
                                      ; zero.
11. NEXT:    CMP     TAIL,HEAD        ; Is TAIL = HEAD?
12.          BEQ     QFULL            ; If so, branch for full queue
                                      ; processing.
13.          MOV     TAIL,TEMP
14.          ASL     TEMP             ; Multiply TAIL by 2.
15.          MOV     AC,QUEUE(TEMP)   ; Add item to queue.
             ...
16. GET:     CMP     HEAD,TAIL        ; Is HEAD = TAIL?
17.          BEQ     QEMPTY           ; If so, branch to QEMPTY.
18.          INC     HEAD             ; Otherwise increment
                                      ; HEAD.
19.          CMP     HEAD,MAX         ; Is HEAD > MAX?
20.          BLE     NEXT1            ; If not, continue.
21.          CLR     HEAD             ; Otherwise clear HEAD to
                                      ; zero.
22. NEXT1:   MOV     HEAD,TEMP
23.          ASL     TEMP             ; Multiply index by 2.
24.          MOV     QUEUE(TEMP),AC   ; Remove item from queue.
```

The following sequence is for System/360-370:

```
     LABEL   CODE   OPERAND          COMMENTS
1.   AC      EQU    3
2.   TAIL    EQU    4
3.   HEAD    EQU    5
4.   TEMP    EQU    6                A temporary register.
5.           SR     TAIL,TAIL        Initialize queue.
6.           SR     HEAD,HEAD
     . . .   . . .  . . .            . . .
7.   PUT     A      TAIL,=F'1'       Add 1 to TAIL.
8.           C      TAIL,MAX         Is TAIL > MAX?
9.           BNH    NEXT             If not, continue.
10.          SR     TAIL,TAIL        Otherwise clear TAIL to zero.
11.  NEXT    CR     TAIL,HEAD        Is TAIL = HEAD?
12.          BE     QFULL
13.          LR     TEMP,TAIL
14.          SLA    TEMP,2           Multiply the index by 4.
15.          ST     AC,QUEUE(TEMP)   Add item to queue.
     . . .   . . .  . . .            . . .
16.  GET     CR     HEAD,TAIL        Is TAIL = HEAD?
17.          BE     QEMPTY
18.          A      HEAD,=F'1'       Add 1 to HEAD.
19.          C      HEAD,MAX         Is HEAD > MAX?
20.          BNH    NEXT1            If not, continue.
21.          SR     HEAD,HEAD        Otherwise clear HEAD to
                                     zero.
22.  NEXT1   LR     TEMP,HEAD
23.          SLA    TEMP,2           Multiply the index by 4.
24.          L      AC,QUEUE(TEMP)   Retrieve item from queue.
```

Note that a small saving in Example 6.6 is possible if MAX is measured in bytes rather than words. If HEAD and TAIL also are measured in bytes from the beginning of the memory area QUEUE, then no conversion from index to bytes (as in lines 13, 14, 22, and 23 of both sequences) is necessary.

Queues are used for buffering input/output data in many programming systems. For example, the process that calls PUT may be a program that reads characters from a teletype keyboard or from a paper-tape

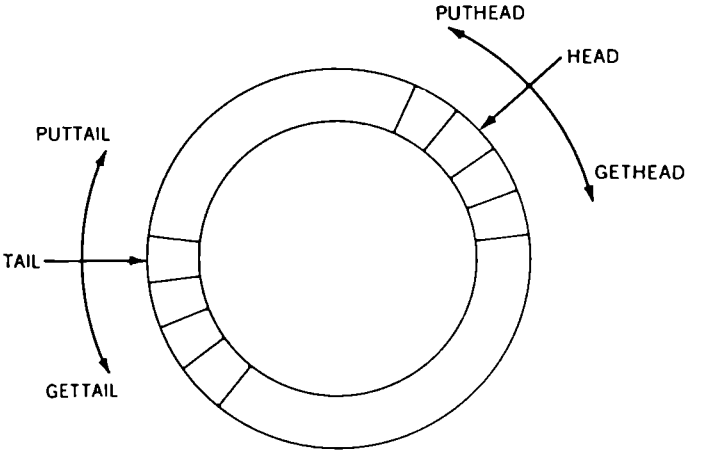


FIGURE 6.1 A double-ended queue

reader. The program that calls GET might be an application program that uses the input characters for data. Queueing is a natural method for permitting an input/output program to operate at the speed of the input/output device, while an application program requests data at a rate determined by the speed of computation of the application program. The input/output program must be prepared to suspend operation when the queue becomes filled until the application program catches up and empties one or more positions in the queue. Similarly, the application program must temporarily suspend operation when the queue becomes empty. This point is discussed more thoroughly in Chapter 8.

There are some algorithms for which it is convenient to generalize the notion of a queue to a data structure for which items may be inserted at either end. Such a data structure is called a *double-ended queue* or a *deque*. The routines for manipulating a double-ended queue are straightforward generalizations of PUT and GET. For convenience let us call these routines PUTHEAD, PUTTAIL, GETHEAD, and GETTAIL. Figure 6.1 illustrates how these routines function. The double-ended queue is shown as a circular collection of data, with the pointers HEAD and TAIL specifying two elements. The curved arrows indicate the movement of the pointers when a routine is invoked.

6.4 PUSH-DOWN STACKS

Push-down stacks are one-dimensional arrays that are like queues in that items are entered and removed one at a time. The difference between push-down stacks and queues is that the retrieval discipline for stacks is last-in-first-out and that for queues is first-in-first-out.

Items are entered and removed from stacks one at a time, so that the last item placed in the stack is the first one to be removed. The behavior of push-down stacks has often been compared to that of an ordinary stack of cafeteria trays.

The process of adding an item to a stack is called a *push-down* or *push*. The process of removing an item is called *pop-up* or *pop*. To implement a push-down stack in machine language, we use an array and a single variable called TOP that gives the index of the current top of the stack. The stack starts at one end of a linear array reserved for it and expands toward the other end as items are added. PUSH and POP routines can be written in the following way in an ALGOL-like language:

```
PUSH:   if TOP ≤ LOWERBOUND then
            go to STACKOVERFLOW
        else
            begin
                TOP ← TOP−1;
                STACK[TOP] ← AC;
            end
POP:    if TOP > UPPERBOUND then
            go to STACKUNDERFLOW
        else
            begin
                AC ← STACK[TOP];
                TOP ← TOP+1;
            end
        . . .
```

The autodecrement addressing mode in the PDP-11 is very useful for implementing stack data structures. Recall that the autodecrement mode is very similar to the autoincrement mode, the only difference being that the register is decremented *before* use rather than incremented *after* use.

Example 6.7 depicts implementations of PUSH and POP using the autoincrement and autodecrement addressing modes. From Example 6.7 note that the use of autoincrement/autodecrement modes leads to a sub-

EXAMPLE 6.7

The first sequence of instructions is for the PDP-11 computer; the second is for System/360-370. Since the autoincrement/autodecrement addressing modes use the contents of a register as a memory

address, elements of the stack are accessed directly by address from the beginning of the stack memory area rather than by index. In the PDP-11 REG[6] is reserved as the stack pointer and is denoted by SP. However, any register can be used as a stack pointer with the autoincrement and autodecrement addressing modes.

```
    LABEL       CODE    OPERAND              COMMENTS
1.  UPPER-              0                    ; Largest address in
    ADDRESS:                                 ; array stored here.
2.  LOWER-              0                    ; Lowest address in
    ADDRESS:                                 ; array stored here.
3.  AC          =       %0
4.  SP          =       %6
5.              MOV     UPPERADDRESS,SP      ; Initialize stack
                                             ; pointer.
6.              ADD     TWO,SP               ; Move SP to point just
                                             ; outside of stack.
    . . .       . . .   . . .                  . . .
7.  PUSH:       CMP     SP,LOWERADDRESS      ; Is SP ≤ LOWER-
                                             ; ADDRESS?
8.              BLE     STACKOVERFLOW        ; If so, branch to error.
9.              MOV     AC,—(SP)             ; Otherwise PUSH.
    . . .       . . .   . . .                  . . .
10. POP:        CMP     SP,UPPERADDRESS      ; Is SP > UPPER-
                                             ; ADDRESS?
11.             BGT     STACKUNDERFLOW       ; If so, branch to error.
12.             MOV     (SP)+,AC             ; Otherwise POP.
```

In the following sequence for the System/360-370 the stack is an array and the top is the index of the current top of the stack.

```
    LABEL       CODE    OPERAND     COMMENTS
1.  UPPER-              0           Upper bound stored
    BOUND                           here.
2.  LOWER-              0           Lower bound stored
    BOUND                           here.
3.  AC          EQU     4
4.  TOP         EQU     5           Index of current top of
                                    the stack.
5.  TEMP        EQU     6           Temporary register.
```

```
     LABEL     CODE    OPERAND              COMMENTS
6.             L       TOP,UPPERBOUND       Initialize stack pointer.
7.   ...       ...     ...                  ...
8.   PUSH      C       TOP,LOWERBOUND       Is TOP ≤ LOWER-
                                            BOUND?
9.             BNH     STACKOVERFLOW        If so, branch to error.
10.            S       TOP,=F'1'            Decrement index.
11.            LR      TEMP,TOP
12.            SLA     TEMP,2               Multiply index by 4.
13.            ST      AC,STACK(TEMP)       Push.
     ...       ...     ...                  ...
14.  POP       C       TOP,UPPERBOUND       Is TOP > UPPER-
                                            BOUND?
15.            BH      STACKUNDERFLOW       If so, branch to error.
16.            LR      TEMP,TOP
17.            SLA     TEMP,2               Multiply index by 4.
18.            L       AC,STACK(TEMP)       Pop.
19.            A       TOP,=F'1'            Increment index.
```

stantial saving in instructions over the use of the index addressing mode. In some cases it is possible to dispense with overflow and underflow checks. These checks are equivalent to the checks used for ordinary arrays to guarantee that indices are within range.

Stacks are useful data structures because many algorithms require that data be filed and retrieved on a last-in-first-out basis. Stacks are frequently used in compilers and in the compiled code of programs written in ALGOL and PL/I.

Figure 6.1 indicates that the double-ended queue is a generalization not only of a queue but of a push-down stack as well. If we use only the routines PUTHEAD and GETHEAD, we have a push-down stack. If we use only the routines PUTTAIL and GETHEAD, we have an ordinary queue.

6.5 PACKED DATA STRUCTURES

In the previous discussions data structures have been formed in terms of multiples of machine words. They can also be formed from *data packets*, which are smaller than machine words. In such cases it is sometimes necessary to store two or more packets in a single word in order to save memory space.

A good example of data packing in the PDP-11 computer is the use

of a single word to store two 8-bit bytes. Recall that the PDP-11 is capable of accessing individual bytes. The PDP-11, like the System/360-370, has a complete set of instructions for byte operations comparable to the full-word instructions we have described so far. The instruction

```
MOVB X,%1
```

places the byte addressed by X in the lower eight bits of REG[1] and copies the sign bit (most significant bit) into the upper eight bits of REG[1].

In general, the byte instructions operate exactly like their full-word counterparts except the operand is an 8-bit byte rather than a 16-bit word. However, when accessing data with the autoincrement or autodecrement addressing modes, a byte instruction causes the register contents to be modified by 1, the length of a byte operand, rather than 2. Symbolically, the byte instructions are derived from their full-word counterpart by appending a B to the full-word instruction code (for example, MOV, MOVB; CMP, CMPB; etc.). The difference between the binary encoding for word and byte instructions is simply that the most significant bit for a byte instruction is 1. Note that not every full-word instruction has a byte instruction counterpart. Check the *PDP-11 Processor Handbook* for complete details.

The description of the PDP-11 processor given in Chapter 4 can be modified as shown below to include the byte instructions.

```
FETCH state:    MAR ← P;
                  .
                  .
                  .
SOURCE state:     .
                  .
                  .
                if I11–10 = 01 then
                    begin comment autoincrement mode;
                        MAR ← REG[I8–6];
                        SOURCE ← MEMORY[. . . ;MAR];
                        if (I15 = 1) AND (I14–12 ≠ 110) AND (I9 = 0) then
                            begin comment byte instruction;
                                REG[I8–6] ← REG[I8–6]+1
                            end
                        else comment word instruction;
                            REG[I8–6] ← REG[I8–6]+2;
                    end
                else
                if I11–10 = 10 then
                    begin comment autodecrement mode;
                        if (I15 = 1) AND (I14–12 ≠ 110) AND (I9 = 0) then
```

```
            begin comment byte instruction;
                REG[I8–6] ← REG[I8–6]–1
            end
        else comment word instruction;
            REG[I8–6] ← REG[I8–6]–2;
        MAR ← REG[I8–6];
        SOURCE ← MEMORY[. . . ;MAR]
    end
else
    .
    .
    .
```

Now consider the case in which each byte contains two 4-bit digits. Let the digit that occupies the most significant bits of a byte be called the *high* digit and the digit that occupies the low-order bits be called the *low* digit. The following example illustrates two sequences of instructions, one for retrieving a high digit and one for storing a low digit.

EXAMPLE 6.8

At the label HGET we assume that REG[1] contains the address of the byte to be accessed. The digit is left in REG[1], right-justified. All the high-order bits of the result word are 0's.

```
    LABEL   CODE    OPERAND     COMMENTS
1.  R1      =       %1
2.  HGET:   MOVB    @R1,R1      ; Fetch byte to REG[1].
3.          ASR     R1          ; Shift right.
4.          ASR     R1
5.          ASR     R1
6.          ASR     R1
7.          BIC     MASK,R1     ; Mask out high-order bits.
    . . .           . . .         . . .
8.  MASK:           177760      ; Mask has 1's in the 12 high-order bit
                                ; positions.
```

At the label LPUT, REG[1] contains the digit to be stored in memory and REG[2] contains the address of the byte to receive the digit. The high-

order digit of the byte in memory is left unchanged when its low-order digit is updated.

```
     LABEL   CODE    OPERAND    COMMENTS
1.   R1      =       %1
2.   R2      =       %2
3.   R3      =       %3
4.   LPUT:   BIC     MASK,R1    Mask out high-order bits.
5.           MOV     MASK,R3    Form a mask to mask out low-order
                                bits.
6.           COM     R3
7.           BICB    R3,@R2     Mask out low-order bits of byte in
                                memory.
8.           BISB    R1,@R2     Insert new low-order character.
             ...
9.   MASK:   177760
```

The routines above illustrate the fundamentals of data packing and unpacking. The usual conventions are the following.

To retrieve packed data:

1. Fetch the word from memory.
2. Isolate the field, using masking or logical-shift operations or a combination of both.
3. Leave the isolated field right-justified in the result register.

To insert a datum into a packed data word:

1. Fetch the word from memory.
2. Assume that the new datum is right-justified. Using a combination of masking and logical-shift operations, place the new datum in the appropriate field of the word. Leave all other portions of the word unchanged.
3. Restore in memory.

Examples 6.9 and 6.10 illustrate the use of masking and logical-shift operations for packing new data into a word of memory.

EXAMPLE 6.9

Consider the action of a program that places a three-bit entry in bits 12 through 14 of a word in memory by using the BIT CLEAR, OR, and logical-shift operations. The data have the following values:

Memory word: 167532_8

Data word: 000003_8

Mask: 070000_8

Use BIT CLEAR to mask memory word: (NOT 070000) AND 167532 = 107532

Shift data word: 000003 left shifted 12 bits = 030000

Complement mask: $\overline{070000} = 107777$

Use BIT CLEAR to mask data word with new mask: (NOT 107777) AND 030000 = 030000

Form logical OR: 030000 OR 107532 = 137532

EXAMPLE 6.10

Repeat Example 6.9 using the AND, OR, and logical-shift operations. Note that the mask is the complement of the mask used in the previous example.

Memory word: 167532_8

Data word: 000003_8

Mask: 107777_8

Mask memory word: 107777 AND 167532 = 107532

Shift data word: 000003 left shifted 12 bits = 030000

Complement mask: $\overline{107777} = 070000$

Mask data word with new mask: 070000 AND 030000 = 030000

Form logical OR: 030000 OR 107532 = 137532

Example 6.9 can be coded directly for the PDP-11 and Example 6.10 for the System/360-370. The System/360-370 logical-shift instructions

also provide added flexibility in that the length and position of a field can be parameters of these instructions that are computed during program execution. Example 6.11 illustrates how to use logical shifts to isolate a specified data field.

EXAMPLE 6.11

This instruction sequence accesses the word whose address is in REG[ADDR] and isolates the n-bit field of the word beginning at bit m. The values of n and m are held in REG[N] and REG[M], respectively. Since there are m bits to the left of the field to be isolated, the SLL instruction left-justifies this field in the odd register of an even-odd register pair. Then the SLDL instruction shifts the field into the previously cleared even register, leaving it right-justified.

	LABEL	CODE	OPERAND	COMMENTS
1.	N	EQU	2	REG[N] contains field length.
2.	M	EQU	3	REG[M] contains position of the leftmost bit of field. Recall that bits in a word are numbered from 0 through 31, starting at left.
3.	EVEN	EQU	4	The even register of an even-odd register pair.
4.	ODD	EQU	EVEN+1	The odd register of a pair.
5.	ADDR	EQU	6	Address of datum is in REG[6].
6.		L	ODD,0(ADDR)	Fetch datum to REG[ODD].
7.		L	EVEN,=F'0'	Clear even register.
8.		SLL	ODD,0(M)	Left-shift m bits, leaving data left-adjusted in REG[ODD].
9.		SLDL	EVEN,0(N)	Left-shift REG[EVEN;ODD], leaving data right-adjusted in REG[EVEN] with high-order bits cleared.

Note in Example 6.11 how the logical-shift instructions have been used to force 0's into the high-order bits of a register in lieu of using an AND operation with a mask. In this particular instance the logical-shift instructions and the flexibility provided by single and double registers led to a more compact instruction sequence than we could have obtained with masking operations. This is not to be interpreted as a general rule, for there are some instances in which masking can be more efficient than shifting. Usually several different programming approaches must be tried

for a few problems in order to gain a feel for techniques that are likely to make the most efficient use of instructions.

EXERCISES

6.1 An upper triangular matrix is defined to be a square matrix with no entries below the main diagonal. That is, if X is a triangular matrix of dimension N, then X[I,J] = 0 for J < I, where $1 \leq$ I, J $\leq N$. Find a way of storing triangular matrices so that no space is reserved for entries below the main diagonal. Indicate the layout of the array in memory and find a formula for the address of X[I,J] in terms of the values of I and J and the base address of X.

6.2 Write a sequence of instructions that generalizes Example 6.5 to nonzero lower bounds and performs checking of subscripts.

6.3 Write sequences of instructions that implement PUT and GET procedures for a finite queue using the autoincrement addressing mode rather than indexing. Compare the results to Example 6.6.

6.4 Examine the PUT and GET routines and determine how many elements are in the queue when it is full.

6.5 Write routines for maintaining a double-ended queue. How many elements are in the queue when it is full?

6.6 Write an assembly language routine that either sets or clears the nth bit of a word. The value of n is assumed to be given to the routine in a machine register, and another register contains 0 if the bit is to be cleared or 1 if the bit is to be set.

6.7 Suppose that you are to create a packed data array in which each element is four bits long. Thus a 16-bit word in the PDP-11 computer holds four array elements and the 32-bit word in the System/360-370 computers holds eight array elements. Assume that there are 64 array elements held in contiguous words of memory. Write a pair of routines for respectively storing and retrieving the ith element of the array. Assume that the value of i is given in a machine register and that the routine for storing the ith element also receives the new value of the ith element in a machine register.

6.8 Augment the PDP-11 simulator of Exercise 4.5 to include byte instructions. To gain further experience with the PDP-11 instruction repertoire, you might try writing the simulator in PDP-11 assembly language.

SEVEN

SUBROUTINES, COROUTINES, AND PARAMETERS

In the previous chapters the major emphasis has been directed to short sequences of instructions for primitive operations. In this chapter we shall discuss how to organize sequences of instructions into self-contained subprograms called *subroutines* and *coroutines*. The concepts embodied in subroutines and coroutines are fundamental to programming. Except for the most trivial programs, virtually every program is written as a collection of subroutines or coroutines, although, in principle, no program has to be organized this way. As we investigate the mechanics of creating them, we shall also see more clearly the reasons why subroutines and coroutines are important to computer programming.

7.1 SUBROUTINES

It is frequently necessary to execute a particular sequence of instructions at several different places in a program. For example, there may be several points at which the sine of an angle is calculated. When we structure programs as subroutines, we need not duplicate instruction sequences in memory; consequently, we can use memory more efficiently.

Subroutines, then, are instruction sequences that can be entered from many different places in a program. When subroutine processing is complete, control returns to the point from which the branch to the subroutine occurred. To achieve this behavior, an address is supplied to a subroutine each time it is entered. The subroutine branches to this address when it completes its computation. Subroutine entries and exits are so common that all computers have special instructions for doing these processes efficiently. We shall first investigate these instructions for the PDP-11 and System/360-370 and then see how they can be used to construct subroutines.

One of the instructions in the repertoire of the PDP-11 computer is the jump to subroutine (code JSR) instruction. The JSR instruction specifies a register and an address as follows:

```
JSR %5,SUBR
```

When the JSR instruction is executed, it computes an effective address for a jump. Recall that the jump instruction (code JMP) is similar to an unconditional branch to the calculated effective address. However, unlike the JMP instruction, the JSR first stores the address of the next consecutive instruction. More specifically, the contents of a register, REG[5] in this example, are pushed down into a stack in memory to make room for the P register, and then the P register is stored in REG[5]. (When the P register is copied into REG[5], it has already been updated to contain the address of the instruction following the JSR instruction.) The instruction automatically uses REG[6] as stack pointer, and there is no way to specify a different stack pointer. Since the source field specifies where to store the P register, any register may be designated as a source register. Memory locations cannot be used in the source field.

The execution details of the JSR %5,SUBR instruction appear below:

```
comment calculate subroutine entry address. Any addressing mode is allowed. Relative address mode is shown for this example;
MAR ← P;
P ← P+2;
MDR ← MEMORY[. . . ;MAR];

comment save the entry address in TMP, an internal machine register;
TMP ← MDR+P;
```

```
comment prepare for push-down by decrementing the stack pointer;
REG[6] ← REG[6]−2;
MAR ← REG[6];

comment save REG[5] on machine stack;
MEMORY[. . . ;MAR] ← REG[5];

comment save P in REG[5];
REG[5] ← P;

comment branch to subroutine;
P ← TMP;
change state to FETCH;
```

The action of saving REG[5] and copying the value of the P register into REG[5] is the unusual aspect of this instruction. Storing the value of P leaves a "trail marker" showing the jump point. The normal jump instruction, JMP X, leaves no such clue; after it is executed, there is no way to tell how instruction X was reached. After execution of JSR %5,SUBR, however, REG[5] contains the address of the instruction following the JSR, and control can be returned to that instruction later. REG[5] is called the *subroutine linkage register*, since it contains a link back to the original program. Note that any register REG[0] through REG[5] can be used for such a subroutine linkage. The BAL and BALR instructions in the System/360-370 repertoire are roughly equivalent to JSR.

With the JSR instruction we can jump from a place in a program, execute a sequence of instructions, and then return to the instruction following the JSR instruction. The return is accomplished by executing a return from subroutine (code RTS) instruction:

```
RTS %5
```

whose effect is to replace P by the contents of REG[5] and replace REG[5] by the contents of the top of the stack. More formally, the computer performs the following:

```
comment reset the program counter;
P ← REG[5];
MAR ← REG[6];
MDR ← MEMORY[. . . ;MAR];

comment restore REG[5] from the push-down stack;
REG[5] ← MDR;

comment increment the stack pointer;
REG[6] ← REG[6]+2;
change state to FETCH;
```

Example 7.1 shows how the JSR and RTS instructions perform a jump to a subroutine and a subsequent return.

EXAMPLE 7.1

The instructions labeled B1 and B2 pass control to the instruction labeled SUB. When the label RETURN is reached, control returns either to C1 or to C2, depending on whether SUB was entered from B1 or from B2.

```
    LABEL      CODE    OPERAND     COMMENTS
    ...        ...     ...         ...
1.  B1:        JSR     %5,SUB      ; Call subroutine SUB.
2.  C1:        MOV     X,AC        ; Return point from SUB.
    ...        ...     ...         ...
3.  B2:        JSR     %5,SUB      ; Call subroutine SUB.
4.  C2:        MOV     Y,AC        ; Return point from SUB.
    ...        ...     ...         ...
5.  SUB:       INC     AC          ; First instruction of SUB.
    ...        ...     ...         ...
6.  RETURN:    RTS     %5          ; The next instruction will be exe-
                                   ; cuted from the address stored in
                                   ; REG[5].
```

Several aspects of Example 7.1 deserve comment. Assume that there are no instructions between SUB and RETURN that branch out of that region. Then the program behaves as if the instructions labeled B1 and B2 had both been replaced by the sequence of instructions beginning at SUB and ending at RETURN.

The sequence of instructions beginning at SUB and ending at RETURN is called a *subroutine*. It can be invoked from many places in a program, and it will pass control back to the instruction following the one that invoked it. Instructions B1 and B2 are said to *call* the subroutine SUB when they are executed. The address stored in REG[5] is called the *return address*. In Example 7.1, the return address can be either C1 or C2, depending on which instruction calls the subroutine.

The instructions labeled B1 and B2 in Example 7.2 branch to the instruction labeled SUB, which is the entry point of a subroutine. Note that the return address is stored in REG[RETURN] before the branch is taken. The instruction labeled EXIT returns to the point from which the subroutine is called by branching to the address that is stored in REG[RETURN].

Some of the important characteristics of subroutines are apparent from Examples 7.1 and 7.2. In particular, there can be a substantial

EXAMPLE 7.2

The following instructions illustrate how subroutine calls can be implemented in a System/360-370 program:

	LABEL	CODE	OPERAND	COMMENTS
1.	RETURN	EQU	14	REG[14] will hold return address.
2.	B1	BAL	RETURN,SUB	Branch to SUB and store return address in REG[RETURN].
3.	C1	L	2,X	
	...	...	...	...
4.	B2	BAL	RETURN,SUB	Branch to SUB and store return address in REG[RETURN].
5.	C2	L	3,Y	
	...	...	...	...
6.	SUB	LR	2,4	First instruction of subroutine.
	...	...	...	...
7.	EXIT	BR	RETURN	Return to calling program.

saving of memory space by using subroutines. Each subroutine call takes only one instruction, but a subroutine call can invoke a very long sequence of instructions. Thus one copy of a subroutine called from many different places in a program can do the work of many copies of the same subroutine. We pay a small penalty for this space saving in that the subroutine call and subroutine return instructions take some execution time that would disappear if a subroutine call were replaced by the body of the subroutine.

Some advantages mentioned previously for macro-instructions apply to subroutines. Programs written as collections of subroutines or macros can be modified easily because of their modularity. If a subroutine is changed in one place, that change is propagated to every place in a program that calls that subroutine. If the same program were written without subroutines or macros, a particular change might have to be made in 10 places, or in some extreme cases, several dozen places. Changes of this magnitude are very susceptible to human error.

Note that a subroutine must not distrub the contents of any register that the calling program expects to remain unchanged. In general, a subroutine must preserve the contents of a register before using it and must restore the contents upon subroutine exit. Predetermined memory locations can be used to temporarily store the contents of registers.

Examination of Example 7.1 reveals that the subroutine SUB may call other subroutines. These can in turn call further subroutines. In fact, subroutine calls can extend any number of levels. The important characteristic is that a subroutine is always entered from its entry point and always returns to its return address. Hence if subroutine SUB calls another subroutine—say, SUB1—SUB1 must eventually return to SUB, which in turn will eventually return to the program that called it. Figure 7.1 shows a typical sequence of subroutine calls.

Figure 7.1 illustrates one of the advantages of the push-down stack for storing return addresses and partially explains why the PDP-11 makes explicit use of this method in its instruction repertoire. In a sequence of subroutine calls, each subroutine call produces a return address to be saved and each subroutine exit recovers a return address, which then no longer needs to be saved. The interesting property of subroutines is that return addresses must be recovered in a last-in-first-out manner. For example, when program execution reaches SUB1 in Figure 7.1, two return addresses are active, one to return to SUB and one to return to the

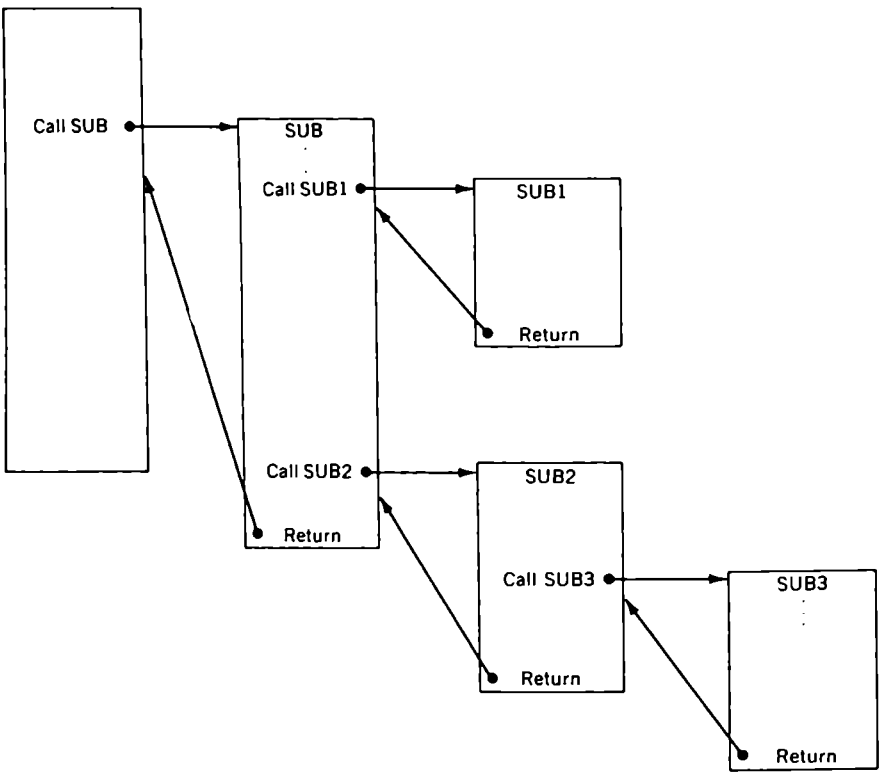


FIGURE 7.1 An example of subroutines that call other subroutines

main program. The exit from SUB1 requires the return address to SUB, which is the most recent return address created. It is quite natural to use a push-down stack to store return addresses because push-down stacks have the last-in-first-out property.

Some computers have instructions that automatically store the return address in a reserved location in the subroutine itself. When this method is used, the act of storing a return address modifies the program, and thus all programs for such computers are self-modifying programs. Programs that do not modify themselves are called *reentrant programs*. By having one copy of a reentrant program in memory, many other programs can make use of it. The current trend in computer design and programming is to avoid program self-modification by isolating program instructions in memory from the data altered by the program. Computers designed specifically to support reentrant programs often can protect reentrant programs from all attempts to write data into themselves; for all such attempts are surely errors. To facilitate the writing of reentrant programs on the System/360-370, the BAL and BALR instructions are designed to leave the return address in a machine register instead of leaving it in a program area. The PDP-11 JSR instruction also supports reentrant programs because the only items altered by this instruction are machine registers and a push-down stack, so no program self-modification need occur. In Section 7.3.1 we return to the question of where to place the return address when we investigate the problem of passing parameters.

7.2 COROUTINES

Coroutines are subprograms that are related to, but more general than, subroutines. Certain computations are better suited to implementation in coroutine form rather than subroutine form, because the subroutine constraints lead to awkward and inefficient programs. In particular, input/output programs and programs that are structured as two or more interdependent processes are often best suited for coroutine implementation.

We can best understand coroutines by comparison to subroutines. A subroutine and the program that calls it behave in a relationship much like that of a supervisor and his surbordinate. The calling routine stops momentarily and requests the subroutine to perform some process. When the subroutine completes operation, the calling routine resumes operation at the point of suspension. Each time a subroutine is called it begins execution from its beginning and terminates at its endpoint.

A pair of coroutines behave more like a pair of coworkers. When coroutine X calls coroutine Y, X supplies a return address, just as in a

subroutine call. The call to Y passes control, not to a beginning point, but to the address supplied by Y when it last called X. Thus coroutines are entered at the instruction that follows their last call, and they do not have a beginning and an endpoint as subroutines have.

Figure 7.2 illustrates how the flow of control passes back and forth between coroutines. This behavior is quite different from the behavior of subroutines. Each time a subroutine is entered, a new computation begins; when the subroutine exit is reached, the computation terminates. A coroutine entry, in contrast, does not necessarily begin a new computation; instead, it resumes the computation that was in progress previously. Consequently, the effect of a coroutine call by coroutine X is to temporarily suspend computation for X, rather than to terminate computation.

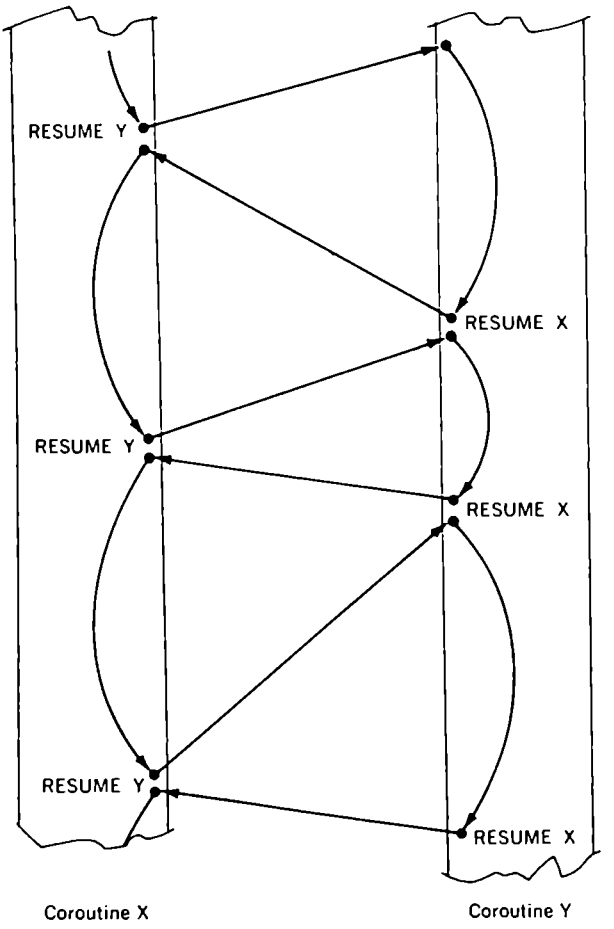


FIGURE 7.2 The flow of control in a sequence of coroutine calls. RESUME denotes a coroutine call. The arrows show the flow of instruction execution

Specifically, a coroutine call consists of the following actions (assume coroutine X calls coroutine Y):

1. Save the value of the P register as the return address for X.
2. Find the return address supplied by Y when it last called X.
3. Enter Y at its last return address.

Example 7.3 shows how to implement coroutines in machine language. We make use of a special form of the JSR instruction to switch from one coroutine to another in the program for the PDP-11 computer. The call takes the form JSR P,@(%6)+. Here the return address for the target coroutine is in the stack and is obtained by the deferred reference @(%6). Since autoincrement mode is used, the stack is popped after the return address is fetched. The JSR generates a new return address and places this in P, the source operand, after the present contents of P are pushed onto the stack. Again REG[6] is used as the stack pointer. But P is itself REG[7], the program counter, so P contains the new return address before it is copied on the push-down stack. Finally, the return address of the target coroutine is placed in P and the target coroutine is reentered. The net effect of JSR P,@(%6)+ is to exchange the contents of the P register with the contents of the top of the stack, and thus an old return address is recovered and replaced with a new one.

EXAMPLE 7.3

The first example is in PDP-11 assembly language, and the second is in System/360-370 assembly language.

```
    LABEL    CODE    OPERAND          COMMENT
1.           MOV     YNAME,—(%6)      ; Initialize stack top to entry ad-
                                      ; dress of coroutine Y. Assume
                                      ; computation started in corou-
                                      ; tine X.
   . . .     . . .   . . .              . . .
2. CALLY:    JSR     P,@(%6)+         ; In coroutine X, a call on Y. The
                                      ; top of the stack and the con-
                                      ; tents of the program counter
                                      ; are exchanged.
   . . .     . . .   . . .              . . .
3. CALLX:    JSR     P,@(%6)+         ; In Y's program, a call on X.
```

For the System/360-370 program we place the return addresses in registers designated by the BALR instructions.

	LABEL	CODE	OPERAND	COMMENT
1.	XRET	EQU	9	Return address for X.
2.	YRET	EQU	10	Return address for Y.
	...	...	...	...
3.	CALLY	BALR	XRET,YRET	In coroutine X, a call on Y. Return address is left in XRET and branch is taken to address in REG[YRET].
	...	...	...	...
4.	CALLX	BALR	YRET,XRET	In Y's program, a call on X.

To illustrate the use of coroutines, let us assume that a main program scans an array of characters from one end to the other and that each word contains one 8-bit character. To conserve space, a string of blanks is stored as one blank followed by an 8-bit integer telling how many blanks are in the string. (If a string of blanks is longer than 255, it is represented as a sequence of strings, each shorter than 255.)

Example 7.4 shows two different programs that do the same computation. The first program uses a subroutine NEXTCHAR to find the next character for the main program. Since each call on NEXTCHAR depends on what happened previously, the first statement of NEXTCHAR determines which portion of NEXTCHAR should be executed. The second program employs coroutines to perform the same process. In this case NEXTCHAR is a coroutine of the main program rather than a subroutine. Note that it is slightly shorter than the corresponding subroutine, and it is easier to comprehend.

EXAMPLE 7.4

We use an ALGOL-like language for the descriptions below. The symbol := is the ALGOL symbol for ← used above. Initialization and stopping conditions are not shown.

comment in this program the characters are stored in the array CHARS, the next item to examine in CHARS is CHARS[I], and subroutine NEXTCHAR places its output in the variable C for the main program to examine. The integer BLANKSREMAINING holds the number of blanks that have to be generated by NEXTCHAR before the next item in CHARS can be examined;

```
subroutine NEXTCHAR;
    begin comment check to see if NEXTCHAR is in the midst of generating
    a string of blanks;
    if BLANKSREMAINING = 0 then
        begin comment find the next item in CHARS and examine it;
            I: = I+1;
            if CHARS[I] ≠ BLANK then
                begin comment output the character to main program;
                    C:= CHARS[I];
                end
            else
                begin comment start generating blanks;
                    I:=I+1:
                    BLANKSREMAINING:=CHARS[I]-1;
                    C:=BLANK;
                end;
        end
    else
        begin comment at this point BLANKSREMAINING is not zero.
        Another blank has to be generated;
                C:=BLANK;
            BLANKSREMAINING:=BLANKSREMAINING-1;
        end
    end subroutine NEXTCHAR;
...
comment the next statement in the main program calls NEXTCHAR to find
the next character;
NEXTCHAR;
comment the next character is in C at this point and is available to the main
program for use;
...
```

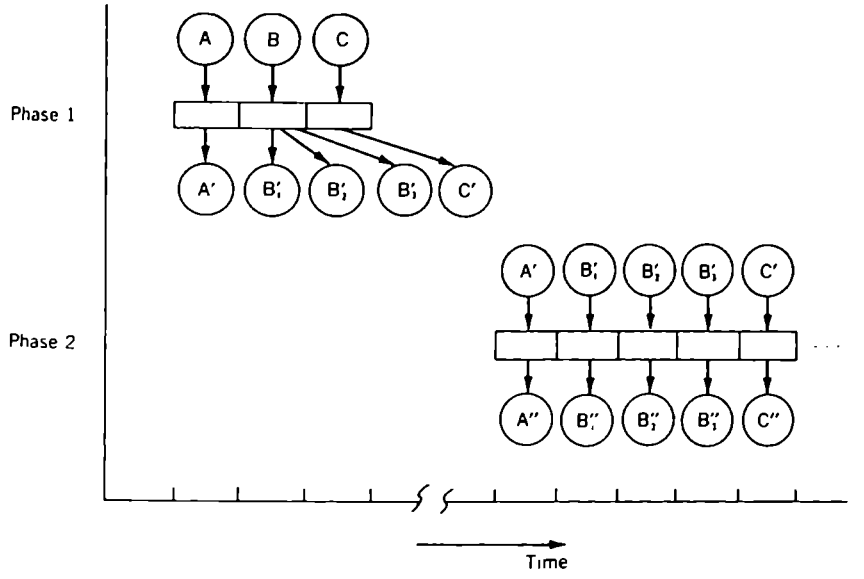
The following is the coroutine version of this program.

```
comment NEXTCHAR and the main program are coroutines. The variables
are the same as in the subroutine version;
coroutine NEXTCHAR;
    begin comment this coroutine is written to loop forever. In real situations
    it should be written to halt when the last character is examined;
    LOOP:  I:=I+1;
        if CHARS[I] ≠ BLANK then
            begin comment the next character is not a blank. It is output to
            the main program;
                C:=CHARS[I];
                RESUME mainprog;
            end
        else
            begin comment generate a string of blanks and output them to
            the main program;
```

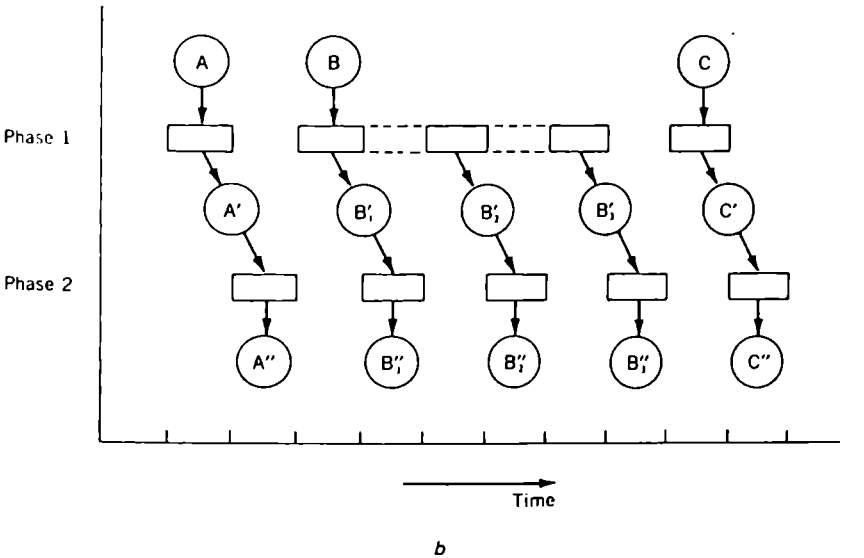
```
                I:=I+1;
                comment CHARS[I] contains the number of blanks to out-
                put;
                for J:=1 step 1 until CHARS[I] do
                    begin C:=BLANK;
                        RESUME mainprog;
                    end
            end;
            go to LOOP;
        end coroutine NEXTCHAR;
    coroutine mainprog;
        ...
            comment the following statement calls NEXTCHAR for the next
            character;
            RESUME NEXTCHAR;
            comment the next character to process is now held in C;
        ...
```

Example 7.4 illustrates an important facet of coroutines. Note that the processing takes place in two phases. In the first phase NEXTCHAR examines each input character and produces a modified version of the input. In the second phase the main program processes the modified input data. The computation alternates between these two phases, so that execution advances in both phases concurrently. The program could also have been structured with the two phases in sequence, that is, with second-phase processing initiated only after the input data had been completely processed by the first phase. In this form the program passes over all the input data twice. In the first pass the NEXTCHAR processing is performed, producing a modified version of the input. In the second pass the main program processes the modified input data. Thus Example 7.4 is equivalent to a two-pass algorithm even though the processing is done in a single pass.

It is usually more efficient to process data in a single pass rather than multiple passes, especially when there is a large amount of data. Consequently, some multiple-pass algorithms can be transformed into more efficient one-pass algorithms in which the different phases of processing are structured as coroutines. Figure 7.3 illustrates this transformation. The horizontal axis is a time axis, so that Figure 7.3*a* depicts a case in which all first-phase processing is completed before second-phase processing is begun. Figure 7.3*b* shows a case in which processing alternates between the two phases. Consider how a typical datum, A, is processed in each program. In the first phase, A is processed and the datum A′ is output, where A and A′ may be different. In the second phase, A′ is changed to A″. Datum B illustrates a case in which new data are gener-



a



b

FIGURE 7.3 Two versions of a multiple-pass computation. Input/output data appear in circles, and each rectangle denotes the execution of a subprogram that processes a single datum. (*a*) The phases are executed in two separate passes of the data and (*b*) program flow alternates between phases

ated during a processing phase. The treatment of B in Figure 7.3 is the same as that of blank characters in Example 7.4.

Coroutines are commonly used for input/output processes as described in the next chapter. Assemblers also use coroutines to process macro-instructions. In principle an assembler could make one pass through a program to expand all macro-instructions, one pass to build a symbol table, and one pass to assemble instructions. The first two passes can be combined if the macro expansion process is a coroutine of the second-pass processor and if all macro-instructions are defined before they are invoked. The second and third passes of an assembler cannot be combined just by structuring them as coroutines because of forward references. When multiple passes are used, forward references can be placed in the symbol table during one pass, ready for use in the next pass. Unfortunately, a coroutine structure causes these two passes to be done concurrently, which raises the difficulty of assembling an address for a symbolic name before the corresponding address is known for that name. In general, multiple-pass processes can be collapsed into a single pass only if the resulting coroutine structure does not create forward-reference problems.

7.3 PARAMETERS

In most instances coroutine calls and subroutine calls include parameters. For example, a call to the sine subroutine is usually accompanied by a number, and the subroutine returns the sine of that number. The return address is also a parameter in a sense, so that subroutine and coroutine calls always transmit at least one parameter. Several techniques for dealing with parameters are in common use, and the characteristics of each should be thoroughly understood by the programmer.

Before we discuss the techniques themselves, let us introduce some terminology. We use the word *parameter* in two distinct contexts, as illustrated in Example 7.5.

EXAMPLE 7.5

The following subroutine is in an ALGOL-like language. The word *procedure* is synonymous with *subroutine.*

```
procedure QUADRATIC(ROOT,A,B,C);
    begin
        ROOT := (—B+SQRT(B↑2—4*A*C))/(2*A);
    end;
```

A typical call on this subroutine is

```
QUADRATIC(ANSWER,1,W,Y);
```

After the call, ANSWER will contain the result.

In this example, ROOT, A, B, and C are dummy variables for the subroutine QUADRATIC. The processing within the subroutine is described in terms of ROOT, A, B, and C, but when the subroutine is executed, these variables will be replaced by other variables. ROOT is replaced by ANSWER, A by 1, etc. Parameters used as dummy variables are called *formal parameters*. The quantities that replace formal parameters when a subroutine is executed are called *actual parameters*. Thus ROOT is a formal parameter, and its corresponding actual parameter is ANSWER.

In considering the various methods of passing parameters between programs two fundamental questions are of concern:

1. Where do we place information that is transmitted to subroutines?
2. What information is transmitted during a subroutine call?

We shall discuss each of these questions in the following subsections.

7.3.1 The Mechanics of Passing Parameters

When we consider methods for transmitting parameters to a subroutine, we wish to minimize both execution time and memory requirements. Several different techniques have come into use; four of them are treated in this section. For any application and any computer at least one of these four should be adequate.

The first technique makes use of *common data*, sometimes known as *global data*. This is nothing more than using memory cells for communication, where the memory cells are accessible to both the subroutine and the program that calls it. For example, a program may include the instructions

```
     ...    ...     ...
     MOV    AC,MAX  ; Store parameter in MAX.
     JSR    RET,Y   ; Call subroutine Y and store return address in
                    ; REG[RET].
     ...    ...     ...
Y:   MOV    MAX,AC  ; Fetch parameter within subroutine Y.
```

When subroutine Y and the main program are assembled together, both routines have access to the same symbolic names, and so both can refer to MAX. When the routines are assembled separately, both can refer to MAX provided we treat MAX as an external symbol. However, external references for many parameters in a large program can become a great inconvenience.

A more fundamental disadvantage is the constraint on addressability imposed by the computer organization. For System/360-370 computers common data cells must be addressable by a base register available both to the main program and to the subroutine. Since the displacement field of a System/360-370 instruction contains 12 bits, a region of memory addressable by a base register is $2^{12} = 4096$ bytes, or 1024 full words. These regions are relatively small, and common data areas are therefore a scarce resource. Another possibility is to use indirect references to MAX, but this solution wastes both memory and execution time.

The second obvious place to put parameters is in the processing-unit registers. At first glance the PDP-11 seems to offer the luxury of using up to eight general registers for passing parameters. However, in reality only a few registers are actually available, because some must be allocated for the program counter, stack pointer, return address, and frequently used temporary values. The same is true for the 16 general registers and 4 floating-point registers of the System/360-370. In general, the more registers we use for parameter passing, the more execution time we can expect to expend for subroutine entry and exit. This is because we often must save the original contents of the registers before we load them with parameters, and the original contents of the registers must be restored when the subroutine processing is completed. In short, processor registers are even more precious than common storage.

A third way of transmitting parameters makes use of special areas of memory called *parameter areas*. Before a subroutine is called, the parameters are placed in successive positions in a parameter area. Then the base address of the parameter area is transmitted to the subroutine in a processor register. The program facing illustrates this technique. In the subroutine the parameters are accessed as elements of the data array by means of techniques already discussed.

A variation of this technique makes use of what are called *in-line parameter areas*. The return address is used as the base address of the parameter area, eliminating the need to transmit an extra address to the subroutine. Naturally, the parameter area must be embedded in the program to do this, and the program will therefore be self-modifying. Subroutine returns for this case are made to a point just beyond the parameter area. Example 7.7 illustrates this technique.

EXAMPLE 7.6

The following is an example of parameter passing for the PDP-11 computer:

```
    LABEL    CODE    OPERAND          COMMENTS
1.  PARAM    =       %1
2.           MOV     PNAME,PARAM      ; REG[PARAM] holds the ad-
                                      ; dress of parameter area.
3.           MOV     ARG1,(PARAM)+    ; Transmit first parameter.
4.           MOV     ARG2,(PARAM)+    ; Second parameter.
             . . .   . . .              . . .
5.           MOV     ARG9,(PARAM)+    ; Last parameter.
6.           JSR     . . .            ; Enter subroutine.
   . . .     . . .   . . .              . . .
7.  PNAME:           PAREA
8.  PAREA: . =       . + 18.          ; Parameter area.
```

There is the same program written for System/360-370:

```
    LABEL    CODE    OPERAND          COMMENTS
1.  TEMP     EQU     1                A temporary register.
2.  PARAM    EQU     10               Register to hold address of
                                      parameter area.
3.           L       PARAM,PNAME      Load address of parameter
                                      area into REG[PARAM].
4.           L       TEMP,ARG1        Pick up first parameter.
5.           ST      0(PARAM)         Store In first word of parameter
                                      area.
6.           L       TEMP,ARG2        Second parameter.
7.           ST      4(PARAM)         Store in second word.
   . . .     . . .   . . .              . . .
8.           L       TEMP,ARG9        Last parameter.
9.           ST      32(PARAM)
10.          BAL     . . .            Enter subroutine.
   . . .     . . .   . . .              . . .
11. PAREA    DS      9F               Parameter area.
12. PNAME    DC      A(PAREA)         Address of parameter area.
```

Note that, in Example 7.7, the subroutine can use the autoincrement addressing mode instead of the index addressing mode to retrieve parameters in the PDP-11 sequence of instructions. Use of the autoincrement addressing mode relieves the programmer of calculating offsets from the base of the parameter area to the parameters as required by the index addressing mode. When, however, parameters are transmitted out of sequence, the index addressing mode is less confusing because the autoincrement mode offsets are more difficult to calculate.

EXAMPLE 7.7

The first sequence is written in PDP-11 assembly language, and the second is in System/360-370 assembly language. The programs show the use of in-line parameter areas.

```
     LABEL     CODE    OPERAND          COMMENTS
1.   PARAM     =       %0
2.   RET       =       %5
3.             MOV     PNAME,PARAM      ; REG[PARAM] holds the
                                        ; address of parameter
                                        ; area.
4.             MOV     ARG1,(PARAM)+    ; Transmit first parameter.
5.             MOV     ARG2,(PARAM)+    ; Second parameter.
               ...
6.             MOV     ARG9,(PARAM)+    ; Last parameter.
7.             JSR     RET,Y            ; Call Y with return
                                        ; address in REG[RET].
8.   PAREA: .  =       . + 18.          ; Parameter area follows
                                        ; JSR.
9.   NEXT:     MOV     Z,T              ; First instruction executed
                                        ; after return from Y.
     ...       ...                        ...
10.  Y:        ...
11.            MOV     4(RET),TEMP      ; Load third parameter
                                        ; into TEMP. REG[RET]
                                        ; contains the starting ad-
                                        ; dress of the parameter
                                        ; area, and the third
                                        ; parameter is four bytes
                                        ; beyond the base of the
                                        ; area.
     ...       ...     ...                ...
```

```
     LABEL      CODE    OPERAND          COMMENTS
12.             ADD     EIGHTEEN,RET     ; Calculate actual return
                                         ; address, nine words be-
                                         ; yond address in
                                         ; REG[RET].
13.             RTS     RET              ; Exit from Y.
14.  PNAME:             PAREA
15.  EIGHTEEN:          18.
```

The System/360-370 version of in-line parameter areas:

```
     LABEL      CODE    OPERAND          COMMENTS
1.   TEMP       EQU     1                A temporary register.
2.   PARAM      EQU     10               Register to hold address of
                                         parameter area.
3.   RET        EQU     14               Register for return address.
     . . .      . . .   . . .            . . .
4.              L       PARAM,PNAME      Load address of parameter
                                         area into REG[PARAM].
5.              L       TEMP,ARG1        Load first parameter.
6.              ST      TEMP,0(PARAM)    Store in first word of param-
                                         eter area.
                . . .   . . .            . . .
7.              L       TEMP,ARG9        Load last parameter.
8.              ST      TEMP,32(PARAM)   Store in ninth word of
                                         parameter area.
9.              BAL     RET,Y            Call Y.
10.  PAREA      DS      9F               Reserve nine full words for
                                         parameter area.
11.  NEXT                                The instruction to which Y
                                         will return.
     . . .      . . .                    . . .
12.  Y          . . .
13.             L       TEMP,8(RET)      Load third parameter.
     . . .      . . .   . . .            . . .
14.  YEXIT      A       RET,=F'36'       Compute return address (in-
                                         crease by nine full words).
15.             BR      RET              Return to calling program.
16.  PNAME      DC      A(PAREA)         Address of parameter area.
```

The last technique, which makes use of push-down stacks, has characteristics that make it desirable in the general case. Parameters are placed in a push-down stack prior to entry to a subroutine. After entry, the subroutine accesses the top cells of the push-down stack for its parameters. Parameters are accessed by indexing relative to the stack pointer. Upon exit the stack is popped up and the parameters disappear, as they should. In the case of a nested sequence of subroutine calls, the last set of parameters to be created is the first to be removed in a sequence of subroutine exits. The last-in-first-out characteristic of parameters matches precisely the same characteristic of push-down stacks. Example 7.8 shows how arguments are passed in a push-down stack.

EXAMPLE 7.8

The first sequence is written in PDP-11 assembly language, and the second is in System/360-370 assembly language.

```
    LABEL   CODE    OPERAND          COMMENTS
1.  POINT   =       %6
2.  RET     =       %5
3.          MOV     ARG2,-(POINT)    ; Push down second param-
                                     ; eter.
4.          MOV     ARG1,-(POINT)    ; Push down first parameter.
5.          JSR     RET,Y            ; Call Y with return address in
            ...                      ; REG[RET].
6.  Y:      MOV     (POINT)+,TEMP    ; Entry to Y. Save old value of
                                     ; REG[RET], now on top of
                                     ; stack, in TEMP.
            ...     ...                ...
7.          MOV     0(POINT),HOLD    ; Load first parameter.
8.          MOV     2(POINT),HOLD1   ; Load second parameter.
            ...     ...                ...
9.          ADD     #4,POINT         ; To return, first pop parame-
                                     ; ters from stack.
10.         MOV     TEMP,-(POINT)    ; Place old value of REG[RET]
                                     ; on stack.
11.         RTS     RET              ; Exit from Y.
```

The System/360-370 version of the same program is as follows. Note that the stack pointer is decremented to push down each parameter, so that parameters can be recovered with positive displacements added to the stack pointer value.

```
     LABEL   CODE   OPERAND          COMMENTS
1.   TEMP    EQU    11               A temporary register.
2.   POINT   EQU    12               Address of current top of stack.
     . . .   . . .  . . .            . . .
3.           S      POINT,=F'4'      Push down second parameter.
4.           L      TEMP,ARG2
5.           ST     TEMP,0(POINT)
6.           S      POINT,=F'4'      Push down first parameter.
7.           L      TEMP,ARG1
8.           ST     TEMP,0(POINT)
9.           BAL    RETURN,Y         Call Y.
     . . .   . . .  . . .            . . .
10.  YTEMP   EQU    7                A temporary register for Y.
11.  Y       L      YTEMP,0(POINT)   Entry to Y. Load first parame-
                                     ter.
             . . .  . . .            . . .
12.          L      YTEMP,4(POINT)   Load second parameter.
             . . .  . . .            . . .
13.          A      POINT,=F'8'      Pop up stack by two words.
14.          BR     RETURN           Return to calling routine.
```

After subroutine entry the first parameter is at the top of the stack, the second next to the top, etc. Hence the parameters are placed in the stack in reverse order. In the PDP-11 program, note that the autodecrement addressing mode greatly simplifies the manipulation of the stack.

One of the major advantages of push-down stacks for parameters is the possibility of using less memory than with other methods. When we create a parameter area for each subroutine by reserving storage ahead of time, as illustrated in Example 7.6, then the total memory reserved for parameters is the total of the parameter areas in the entire program. However, when we use a stack, the total memory reserved in the stack need not exceed the maximum number of parameters that can occur in a

nested sequence of subroutine calls. For some programs the memory saving is considerable.

7.3.2 Call by Name and Call by Value

The previous section deals with the problem of where to place parameters when they are passed to subroutines. In this section we focus attention on what information is actually transmitted. We shall also treat the problem of returning information from subroutines to the programs that call them.

Examination of Example 7.5 reveals that ROOT is used differently from A, B, and C. ROOT receives the output of the computation, whereas A, B, and C are unchanged by the computation. In any assembly language version of Example 7.5 the memory address of the actual parameter that corresponds to ROOT must be supplied to the subroutine, whereas the actual parameters for A, B, and C can be either the memory addresses or the values of the actual parameters. Obviously, if the calling routine transmits addresses to QUADRATIC, then QUADRATIC must be written to receive the addresses rather than the values of the actual parameters.

From this discussion we see that we have two choices for treating parameters A, B, and C, but we must transmit an address for parameter ROOT. Every variable has two attributes that can be used in a program: a memory address and the value contained in that memory address. These are generally referred to as the *name* and the *value* of the variable, respectively. In a high-level language program, when a variable appears in a statement in which it is assigned a value, the execution of the statement uses the name of the variable. Thus the subroutine in Example 7.5 uses the name of ROOT, not the value of ROOT. When a variable appears in an arithmetic expression, its value enters the calculation. If the name of a variable is given, it is possible to find the value of the variable. However, it is not possible, in general, to determine the name from the value.

We can implement subroutine calls by transmitting either the name or the value of a variable to the subroutine. Of course, we can transmit the value of a parameter only if the subroutine does not require the name of the parameter. Let us refer to these two methods as *call by name* and *call by value*. Unfortunately, there is much confusion over these terms in the computer literature, because the programming languages ALGOL, FORTRAN, and PL/I use slightly different definitions. The following discussion should clarify some of the confusion and also bring to light the relative merits of the various methods.

There should be no confusion of terms when the actual parameter is a nonsubscripted variable, because the name and value of such a variable

are well defined. The difficulty arises when the actual parameter is an expression or a subscripted variable. Consider, for example, the following subroutine, which exchanges the values of R and S:

```
procedure SWAP(R,S);
    real R,S;
    begin TEMP := R;
        R := S;
        S := TEMP;
end;
```

Suppose we call SWAP in the following way:

```
X[1] := 2;
X[2] := 5;
I := 1;
SWAP(I,X[I]);
```

If we transmit the addresses of the actual parameters to SWAP, then the values of X[I] and I will be exchanged, which is the intent of the subroutine. However, another interpretation of call by name is possible in this case. In ALGOL the effect of call by name is precisely the same as would be obtained if the actual parameters were substituted for their corresponding formal parameters in the text of the subroutine. This is known as the *replacement rule.* When we apply it here, we obtain the following subroutine:

```
begin TEMP := I;
    I := X[I];
    X[I] := TEMP;
end;
```

Now the second line changes the value of I, so that X[I] in the third line is not the same as X[I] in the second line. For the example given above, the old value of I is assigned to X[2], not to X[1].

The replacement rule adds an entirely new dimension to the notion of call by name. Note that there is no problem when simple variables are used as actual parameters, as in the call SWAP(I,J). The problem arises because the parameter X[I] is an element of an array. The particular element depends on the value of I; in fact, both the name and value of X[I] depend on I.

The replacement rule can lead to difficulties of interpretation. For example, what is the meaning of the call SWAP(5+N,M)? If we adhere

strictly to the replacement rule, we have to interpret the statement

```
5+N := M
```

This statement makes no sense as it stands. The effect during execution depends on the language in which the statement is embedded and can vary among different translators for the same language.

To begin our discussion of subroutine calls let us consider how to implement such calls in assembly language. First we deal with call by value.

ALGOL Call by Value Place the value of the actual parameter in the parameter area. The subroutine accesses only the copy of the parameter in the parameter area. If the subroutine changes this value, the change is not transmitted to the calling routine.

We can use the techniques discussed in the previous section for transmitting parameters to a subroutine. When a parameter is called by value, the information transmitted is the value rather than the name of the parameter. Technically speaking, when an array is called by value, the contents of the array should be copied into a temporary array, which is then transmitted as a parameter. This permits the subroutine to modify the copy of the array without changing the original. Obviously, value calls with array parameters can be grossly inefficient, and some ALGOL compilers protect the programmer by forbidding such calls.

Some FORTRAN compilers, most notably the FORTRAN compilers for System/360-370 computers, use another method of passing parameters that is also known as call by value. In this method, changes in parameters are transmitted back to the calling program at the completion of subroutine execution. Otherwise, this method is identical to ALGOL call by value.

FORTRAN Call by Value Place the value of the actual parameter in the parameter area. The subroutine accesses only the copy of the parameter in the parameter area, possibly changing its value. When subroutine processing is complete, the value in the parameter area is copied back into the original parameter.

This type of call is implemented in the same way as ALGOL call by value, except for processing at the return point in the calling program. At the return point the calling program copies the parameters back into their original locations, which is simply a reversal of the parameter-passing process.

Call by name is somewhat more complex than call by value, as indicated in the discussion of the ALGOL call-by-name replacement rule. Before we discuss the implementation of the replacement rule, let us consider a less general, but more efficient, variation of call by name that

is sometimes known as *call by reference.* This is used in many PL/1 and FORTRAN compilers. The actions for call by reference are as follows.

Call by Reference If the actual parameter is an unsubscripted variable, then place the address of the variable in the parameter area. If the parameter is a subscripted variable, then compute its address and place this in the parameter area. If the parameter is an expression, then compute the value of the expression, store the value in a memory cell, and place the address of this cell in the parameter area.

In every case the subroutine receives the address of the value of the actual parameter. If the subroutine modifies a parameter called by reference, then the original parameter is modified automatically.

As an example, the subroutine call SWAP(I,J) is implemented by transmitting the address of I and J to SWAP. Clearly, when the actual parameter is an unsubscripted variable, call by reference and the ALGOL replacement rule produce identical results. Differences are observed when the actual parameter is a subscripted variable or an arithmetic expression, as we have mentioned earlier.

The implementation of the ALGOL replacement rule requires some discussion. We cannot transmit either the name or the value of an actual parameter to a subroutine for this type of call, because both can be altered during execution of the subroutine. Consequently, we must be able to compute both the name and value of an actual parameter whenever the corresponding formal parameter is referenced. We use the following strategy.

ALGOL Call by Name (Replacement Rule) If the actual parameter is a variable, whether it is subscripted or not, construct a subroutine to compute the name of the parameter and transmit the entry point of this subroutine as the actual parameter. If the parameter is an expression, construct a subroutine that computes the expression, stores it in a memory cell, and produces the name of the memory cell as its output. Transmit the entry point of this subroutine as the actual parameter.

Note that the parameter that is transferred is neither the name nor the value of an entity, but the address of a subroutine that can compute the parameter. To compute the name or value of the parameter, we enter the subroutine supplied; upon exit we have the name of the parameter. Obviously, we can find the value of an item when we are given its name.

The implementation of ALGOL call by name is shown in Example 7.9, which illustrates what happens for the call SWAP(I,X[I]). With this form of call, what happens during the call SWAP(5+N,I)? The first parameter for this call is a subroutine that computes 5+N and stores it in a memory cell, say, at MEMORY[XPRS]. Thus when the replacement rule places 5+N on the left-hand side of the assignment statement, its name is XPRS, and MEMORY[XPRS] receives the value of the assignment.

EXAMPLE 7.9

The following procedure for constructing the call SWAP(I,X[I]) employs ALGOL call-by-name conventions. The first program is in PDP-11 assembly language, and the second is System/360-370 assembly language. In the program for the PDP-11, the subroutines IP and XIP leave the name of the parameter in REG[NAM]. The instructions at RS compute the statement R := S in SWAP.

```
    LABEL    CODE    OPERAND        COMMENTS
    ...      ...     ...              ...
1.  NAM      =       %3             ; Register for names.
2.  RET      =       %5
3.  INAME:           I              ; The name of I.
4.  XNAME:           X              ; The name of X.
5.  I:               0              ; The integer I.
6.  X: .     =       . + 200        ; The array X.
7.  IP:      MOV     INAME,NAM      ; Entry point for subroutine that
                                    ; computes name of I. Puts name
                                    ; of I in REG[NAM].
8.           RTS     RET            ; Return.
9.  XIP:     MOV     I,NAM          ; Entry point for the subroutine
                                    ; that computes name of X[I].
10.          ASL     NAM            ; Make I into an index by multi-
                                    ; plying by 2.
11.          ADD     XNAME,NAM      ; Name of X[I] is in REG[NAM].
12.          RTS     RET            ; Return.
13. IPN:             IP             ; Address of entry point IP.
14. XIPN:            XIP            ; Address of entry point XIP.
    ...      ...     ...              ...
15. CALL:    MOV     IPN,%0         ; First parameter to REG[0].
16.          MOV     XIPN,%1        ; Second parameter to REG[1].
17.          JSR     RET,SWAP       ; Call SWAP.
    ...      ...     ...              ...
18. P1:              0              ; Save space for first parameter.
19. P2:              0              ; Space for second parameter.
    ...      ...     ...              ...
20. SWAP:    MOV     %0,P1          ; Save first parameter.
```

```
     LABEL    CODE    OPERAND      COMMENTS
21.           MOV     %1,P2        ; Save second parameter.
      ...     ...     ...            ...
                                   ; The following four lines show
                                   ; the computation of R := S,
                                   ; where R and S are formal
                                   ; parameters.
24.  RS:      JSR     RET,@P2      ; Compute name of second pa-
                                   ; rameter and leave in
                                   ; REG[NAM].
25.           MOV     @NAM,%0      ; Load value of second parame-
                                   ; ter into REG[0].
26.           JSR     RET,@P1      ; Compute name of first param-
                                   ; eter and leave in REG[NAM].
27.           MOV     %0,@NAM      ; R := S.
      ...     ...     ...            ...
28.           RTS     RET          ; Return from SWAP.
```

In the following program for System/360-370, the subroutines IP and XIP leave the value of the parameter in REG[VAL] and the name of the parameter in REG[NAM].

```
     LABEL    CODE    OPERAND      COMMENTS
      ...     ...     ...            ...
1.   RET      EQU     14           Register for return address.
2.   RET1     EQU     13           Register for SWAP's return ad-
                                   dresses.
3.   I        DS      F            Reserve a full word for I.
4.   INAME    DC      A(I)         The name of I.
5.   XNAME    DC      A(X)         The name of X.
6.   X        DS      100F         Reserve 100 full words for array
                                   X.
7.   S        EQU     9            A temporary register for S.
8.   TEM      EQU     10           A temporary register.
9.   NAM      EQU     11           Register for names.
10.  VAL      EQU     12           Register for values.
11.  IP       L       NAM,INAME    Load name of I into REG[NAM].
12.           L       VAL,0(NAM)   Load value into REG[VAL].
13.           BR      RET          Return.
14.  XIP      L       TEM,I        Load I into REG(TEM).
15.           SLA     TEM,2        Make into an index by multiplying
                                   by 4.
```

```
     LABEL   CODE   OPERAND        COMMENTS
16.          L      NAM,XNAME      Load name of X into REG[NAM].
17.          AR     NAM,TEM        Compute name of X[I].
18.          L      VAL,0(NAM)     Compute value of X[I].
19.          BR     RET            Return.
20.  IPN     DC     A(IP)          The name of entry point IP.
21.  XIPN    DC     A(XIP)         The name of entry point XIP.
     . . .   . . .    . . .        . . .
22.  CALL    L      0,IPN          First parameter to REG[0].
23.          L      1,XIPN         Second parameter to REG[1].
24.          BAL    RET1,SWAP      Call SWAP.
     . . .   . . .    . . .        . . .
25.  P1      DS     F              Storage for first parameter.
26.  P2      DS     F              Storage for second parameter.
27.  SWAP    ST     0,P1           Save first parameter.
28.          ST     1,P2           Save second parameter.
                    . . .          The next six instructions do
                                   R := S.
29.  RS      L      TEM,P2         Load address of second param-
                                   eter subroutine.
30.          BALR   RET,TEM        Place value of S into REG[VAL].
31.          LR     S,VAL          Save this value in REG[S].
32.          L      TEM,P1         Load address of first parameter
                                   subroutine.
33.          BALR   RET,TEM        Place name of R into REG[NAM].
34.          ST     S,0(NAM)       R := S.
             . . .    . . .        . . .
35.          BR     RET1           Return from SWAP.
```

In ALGOL each formal parameter is separately specified as a name parameter or a value parameter. Thus an ALGOL program with both a name and a value parameter would appear as:

```
procedure SWAP(A,B);
    value A;
    real A,B;
    begin . . .
```

Note that A is explicitly specified as a value parameter; B, by default, is a name parameter.

Of the various methods available for transmitting parameters, call

by reference is the one that usually requires the least execution time and is the most widely applicable. The ALGOL methodology probably encompasses the most general cases, but the cost in execution time and memory space can be excessive.

The different techniques for transmitting parameters to subroutines are essentially a set of tools for the computer programmer. A thorough knowledge of the techniques should provide greater understanding of the processes that take place during the execution of programs written in high-level languages. From the standpoint of program design, it is essential that the programmer be familiar with the various alternatives, that he know their relative merits, and that he be especially aware of their pitfalls.

REFERENCES

The concept of subroutines has been a part of the literature of algorithms for many years. Early Turing machine programs made much use of subroutines, as did instruction sequences for other abstract mathematical models of computers. Consequently, instructions for implementing subroutine calls were part of the repertoire of all the early digital computers. The coroutine, however, is a later invention, an outgrowth of the notion of generators embodied in the list-processing languages of Simon, Newell, and Shaw (Newell, 1961). The term *coroutine* is credited to Conway, who has given a particularly lucid description of this concept (Conway, 1963). Techniques described here for parameter passing are representative of several compiler implementations and are in common use. The use of the push-down stacks as a parameter area appeared in the latter part of the 1950s, again in the languages designed by Simon, Newell, and Shaw. Much of the interest in push-down stacks was stimulated by ALGOL and by the list-processing languages, because the push-down stack is especially appropriate for these languages. The subroutines employed in the technique for implementing the ALGOL replacement rules are sometimes called "thunks" and have been described by Ingerman (1961).

EXERCISES

7.1 Show how an assembly language program in which each subroutine call places its return address in the first location of the subroutine

can be written. Indicate how the return address is placed in the memory location and how it is used during a subroutine exit.

7.2 Suppose that during the execution of a program the subroutine SINE for some input data calls the subroutine COSINE, which in turn might call SINE. Show that no difficulty occurs if the return addresses are put into push-down stacks but that if another nonequivalent method is used for return addresses, the program may loop forever.

7.3 Show the assembly language for the coroutine calls and exists for Example 7.4.

7.4 The program below is written in a nonstandard programming language. When it is executed, the printed output contains the number 5. Explain how this can happen.

```
begin comment the main program;
    i ← 3;
    call increase (i,2);
    i ← 2;
    print i;
end main program;
subroutine increase(a,b);
    begin b ← b+a;
    end;
```

EIGHT

INPUT/OUTPUT OPERATIONS

All computers have some mechanism for exchanging information with the outside world. The information may be a program or input data for a program or the results of a computation. The outside world might gain access to the computer through such peripheral devices as typewriters, line printers, punched-card readers, paper-tape readers and punches, or push-button telephones. If a computer has an external auxiliary memory, such as a magnetic tape or disk, the auxiliary memory can be considered part of the outside world, and information can be exchanged in the form of library programs or large data files. In any case, input/output processing is one of the central problems of computer programming.

Input/output programs are among the most difficult to write for a number of reasons. Because input/output actions are performed simultaneously with computation, timing factors can cause programs to fail in subtle ways. It is also typical of most computer systems that each peripheral device must be treated differently, so that there must be a program for each device rather than one program for all devices.

In this chapter we shall discuss programming techniques for both the PDP-11 computer and the IBM System/360-370 computers. Normally, the programmer does not prepare input/output programs for either of these computers; the program libraries contain a complete collection of the necessary input/output subroutines. In fact, the programmer is specifically denied access to the input/output instructions of System/360-370 computers, and these instructions can be executed only by an executive program.

Unfortunately, it is somewhat impractical to describe the input output libraries for the PDP-11 or System/360-370 computers. Such libraries tend to be tailored to the specific requirements of each computer installation, with correspondingly tailored program conventions and parameters for library subroutines. Hence it is up to the individual programmer to inform himself of the characteristics of the input/output library for his particular installation. Nevertheless, a thorough understanding of the material discussed in this chapter should assist the programmer in making use of input/output libraries and enable him to obtain maximum performance from a computer.

8.1 FUNDAMENTALS

Each input/output operation involves the simultaneous operation of two different processors. One is the processing unit of the computer, and the other is a special control unit, called an *interface*, that transfers data between computer and peripheral device. Both these processors can operate simultaneously, but they must operate in a nonconflicting way.

The usual mode of operation is to use REQUEST and REPLY control signals. In this mode the computer sends a REQUEST to an input/output device and then proceeds with program computation. When the input/output operation is complete, a REPLY is returned to the computer. The computer, acting under program control, can sense the REPLY signal. If the computer reaches a point at which computation cannot continue until a REPLY is received, it can enter a loop, which it will exit only when the REPLY is received. This is shown graphically in Figure 8.1. The dashed portion of the processor activity line shows the period during which the processor loops while awaiting a REPLY. The dashed line for the interface unit indicates the time during which the interface is idle while waiting for a new REQUEST. Note that both the interface and the processor can be busy simultaneously.

In the PDP-11 computer there are two memory words associated

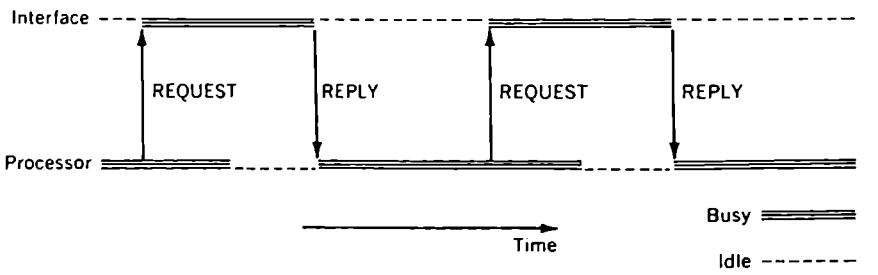


FIGURE 8.1 An example of REQUEST and REPLY signals for controlling input/output operations. Note that the interface and the processor can operate concurrently

with each peripheral device. These are shown in Figure 8.2. The status register is used to record the status of the REQUEST and REPLY signals. The buffer register is used for holding information in transit between the processor and the peripheral device. The interface for a PDP-11 peripheral device is considered part of the peripheral and is not shown separately in the figure. In reality, the interface may be a physically separate module.

The loading of a datum into the buffer register initiates an output operation. When the output operation has completed, a bit (READY) is set to 1 in the status register corresponding to the device.

Input operations are essentially the same as output operations, except that the input data flow is in the opposite direction from the output data flow. An input operation occurs when a REQUEST is transmitted to an input device. This is done by setting an ENABLE bit to 1 in the status register. A BUSY bit is set to 1 in the status register while an input operation is in progress. During this time the processor should not read from or write into the buffer. When the input operation is complete, the device becomes inactive and issues a REPLY. The REPLY is recorded by setting the DONE bit to 1 and the BUSY bit to 0 in the status register. At this point the processor can remove the data from the buffer register and initiate a new REQUEST.

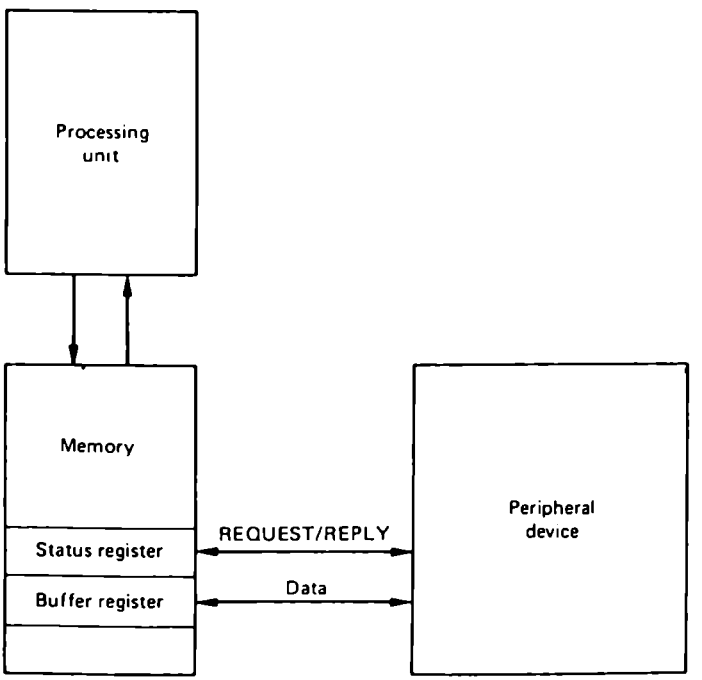


FIGURE 8.2 A diagram of the PDP-11 input/output control signals for one peripheral device

The relation of the READY, ENABLE, and DONE bits to the REQUEST and REPLY signals is indicated by Table 8.1. Figure 8.3 depicts the status register and data buffer pair for a typical input and output device. The status register and buffer register have addresses similar to words in memory, although they are not physically part of memory. Thus the PDP-11 instruction set can be used to control input/output operations. Most computers have a separate set of instructions for input/output.

TABLE 8.1

	ACTION OF CONTROL SIGNAL	
SIGNAL	INPUT	OUTPUT
REQUEST	Set ENABLE bit for peripheral device (DONE bit is automatically reset to 0)	Move output datum into buffer register of peripheral device (READY bit is automatically reset to 0)
REPLY	Peripheral device sets DONE bit in status register (Peripheral device will have reset the ENABLE bit to 0 by this time)	Peripheral device sets READY bit to 1 in status register.

The shaded bits in Figure 8.3 are not used. However, some peripheral devices require more control and status information, so that the shaded bits of the status register have some significance for these devices. Note that the basic input/output datum is a byte.

The computer can test for the REPLY from the peripheral device by sensing the READY/DONE bit. It is quite common to use an instruction sequence of the form:

```
        ...
LOOP:   TSTB    DEVICE
        BPL     LOOP
        ...
```

The test instruction (code TST) sets the Z and N condition codes according to the value of its operand. DEVICE is assumed to be the address of the peripheral device status register. Since the READY/DONE bit is the sign bit of the lower byte of the status register, it can be tested by the byte version of TST. Note that the computer is idle until a REPLY is received from the device. This type of loop is called a *wait loop.*

The function of the REQUEST/REPLY signals varies from one device

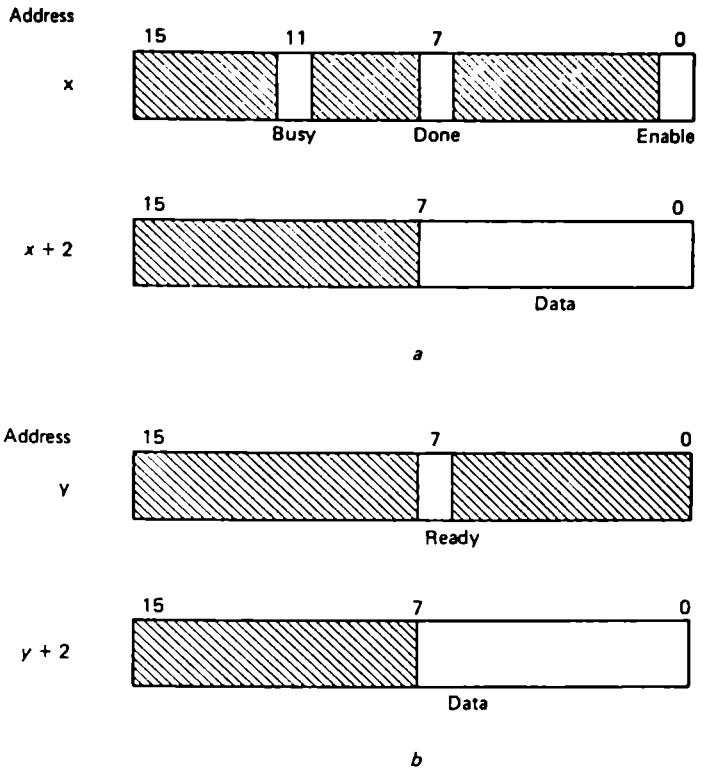


FIGURE 8.3 Typical status register and data buffer words for (*a*) an input device and (*b*) an output device in the PDP-11 computer

to another, so that input/output operations must be described separately for each device. The succeeding sections show basic techniques for the paper-tape reader, the paper-tape punch, and the teletype.

8.1.1 Paper-Tape Reader

A read operation is initiated by the instruction

```
INC DEVICE
```

which sets the reader enable bit. A test should be made to see if the paper-tape reader is busy before using this instruction, unless it is certain the device is idle. The test is made simply by examining the value of bit 11 in the paper-tape-reader status register.

By setting the ENABLE bit, a REQUEST signal is generated to the paper-tape reader, and the BUSY bit is set to 1. A character is read and placed into the buffer register. After the character is placed into the buffer, the REPLY signal resets the BUSY bit to 0 and the DONE bit to 1.

EXAMPLE 8.1

The following program reads one character from paper tape and places the result in the low-order character, bits 0 through 7, of REG[0]. READ is the address of the paper-tape reader status register.

```
LABEL   CODE    OPERAND       COMMENTS
        INC     READ          ; Generate REQUEST to paper-tape
                              ; reader.
...     ...     ...             ...
LOOP:   TSTB    READ          ; Test for completion of READ operation,
                              ; loop if not done.
        BPL     LOOP
        MOVB    READ+2,%0     ; Move data in buffer to REG[0].
```

We obtain maximum efficiency when the reading operation is overlapped with computation as much as possible. Hence a subroutine for reading paper tape should issue an INC READ instruction just before it exits. This causes the next input operation to begin while computation proceeds in the main program. When the READ subroutine is reentered the next time, it should begin with the TSTB instruction and wait for the read operation to finish. Note that for this case one read operation must be initiated before the READ subroutine is entered for the first time. The first read operation can be started by an initialization subroutine that is called only once during program execution.

8.1.2 Paper-Tape Punch

The punch operation is initiated by placing a character in the punch buffer register. When the output operation is complete, the READY bit in the status register is set to 1. Thus a typical sequence of output instructions is the following, where PUNCH is the address of the paper-tape punch status register.

```
LABEL   CODE    OPERAND       COMMENTS
LOOP:   TSTB    PUNCH         ; Wait until unit is ready.
        BPL     LOOP
        MOVB    %2,PUNCH+2    ; Move data from REG[2] to punch
                              ; buffer register.
```

The low-order character, bits 0 to 7 of REG[2], is transferred to the buffer by the MOVB instruction.

For the maximum overlap of output activity with computation, an output subroutine should initiate the output action just before exit. Upon entry to the output subroutine, the first instruction should test the READY bit of the output device and wait for the operation to complete by looping as shown. Hence the sequence of instructions given above forms the body of an output subroutine.

8.1.3 Teletype Input/Output

The teletype is both an input and an output device. It consists of a keyboard and a printing mechanism. These two elements are not mechanically coupled, as they are in a typewriter. Striking a key on the keyboard sends a character to the computer, but it does not cause the print mechanism to operate. The teletype can function like a typewriter, however, if a computer program "listens" to the teletype keyboard and "echoes" the data back to the teletype printer. A typical teletype input program is shown in the following example. Note that in Example 8.2 the input device does not have to be enabled to read a character, since the human operator presses the keys on the teletype keyboard to transmit data.

From the sample teletype input program of Example 8.2 it is straightforward to construct a teletype output program. The following program echoes the teletype keyboard to the teletype printer. TTIN and TTOUT are the addresses of the teletype input and output status registers, respectively.

EXAMPLE 8.2

```
LABEL   CODE    OPERAND     COMMENTS
LOOP:   TSTB    TTIN        ; Wait for character from keyboard.
        BPL     LOOP
        MOVB    TTIN+2,%0   ; Move character to REG[0].
```

Characters entered via the keyboard could be lost if they are entered faster than the print mechanism can respond. The failure occurs when the program of Example 8.3 is in LOOP2 awaiting a chance to echo a character while one or more characters enter the keyboard buffer.

Although the relative speed of the input and output devices in this

EXAMPLE 8.3

```
LABEL   CODE    OPERAND           COMMENTS
LOOP:   TSTB    TTIN              ; Wait for character from key-
                                  ; board.
        BPL     LOOP
LOOP2:  TSTB    TTOUT             ; Wait for printing mechanism to
                                  ; finish.
        BPL     LOOP2
        MOVB    TTIN+2,TTOUT+2    ; Echo keyboard.
        BR      LOOP
```

case allows the computer to keep ahead of the input, there are cases when bursts of input may come faster than the computer can process them. In such cases it is wise to store the input data without taking time to process them until the burst is over. Then the computer can process the input data without danger of losing characters. If the input burst lasts too long, the computer will run out of available memory for holding the data, and some data will be lost. Later in this chapter we shall see that the elapsed time between the execution of critical instructions will make the difference between whether data are lost or not. In Example 8.3 the critical elapsed time is between the successive executions of the MOVB TTIN+2,TTOUT+2 instruction. Timing constraints are more difficult to satisfy when the elapsed time between critical instructions is unpredictable. When we discuss interrupt processing, we shall see that timing unpredictability in programs is the norm rather than the exception.

8.2 INTERRUPTS

The programs in the previous section generally take the following form:

1. Initiate an operation.
2. Compute as long as possible.
3. Wait for the REPLY signal with a TSTB instruction.
4. When ready, go to step 1 and repeat.

However, the third step in this process, the loop containing a TSTB instruction, is often undesirable. While the computer is waiting, it might be capable of performing other operations. When two or more devices

are active, it is essential that the computer be capable of reacting to the first device that becomes ready, instead of idling in a wait loop associated with a single device. Consequently, most computers use a special mechanism called an *interrupt system* for efficient input/output.

The purpose of the interrupt system is to provide for useful computation in place of wait loops. The interrupt system permits the computer to respond to REPLY signals for devices as soon as they are received, without having to enter a wait loop. Specifically, a REPLY from a device can interrupt normal processing and force the computer to execute instructions that control that device. The previous input/output examples show input/output operations when the interrupt system is turned off, since the only effect of the REPLY signals in these cases is to set the corresponding READY/DONE bit to 1. When an interrupt occurs, a return address is saved and a jump is initiated in a manner analogous to the execution of a JSR instruction. Thus we can treat interrupts just like subroutine and coroutine calls. For example, assume that a matrix-inversion program is in execution when an interrupt occurs. At this point the control transfers to the subprogram for the input/output device that generated the interrupt. The input/output program presumably initiates a new operation and then uses the return address back to the matrix-inversion program. This input/output program is essentially a subroutine or coroutine of the matrix-inversion program. However, the matrix-inversion program contains no instructions that call the input/output program explicitly, since the input/output program is called by an interrupt. In this section we shall see how to do input/output operations under interrupt control.

The INTERRUPT state is entered after the EXECUTE state. Since interrupts are recognized only in the INTERRUPT state, the P register at the time of interruption has the address of the next instruction to be executed in the interrupted program. Thus if there is an interrupt, the P register contains the desired return address. The condition codes also must be saved so that the interrupt program can use them and the original codes can be restored when interrupt processing is complete. If a program is interrupted between a CMP and BPL instruction, for example, the condition code settings of the CMP instruction have to be saved and restored so that the BPL instruction can react to them when the program continues.

We will now trace the details of the PDP-11's response to an interrupt. The status register for each device has an interrupt-enable bit (bit 6). When the interrupt-enable bit is on, the setting of the READY/DONE bit by the device causes an interrupt to occur. The interrupt causes the computer to push the contents of the *processor status* (PS) *register*, which contains the condition codes, and the contents of the program counter (P) onto the PDP-11 push-down stack. The PDP-11 processor status

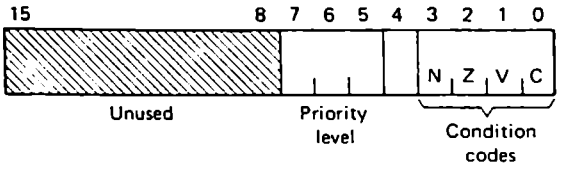


FIGURE 8.4 The PDP-11 processor status register

register is similar to the System/360-370 program status word; it is depicted in Figure 8.4. The condition codes reside in bits 0 through 3. The purpose of bits 5 through 7, the priority level, is described in Section 8.3. The push-down stack is the same stack used by the jump subroutine (JSR) and subroutine return (RTS) instructions. REG[6] is used as the stack pointer. The contents of the PS and P registers are referred to as the *status* of the machine.

After saving the old machine status, the computer uses an address supplied by the interrupting device to fetch a pair of words from memory. The word at the given address is placed in the program counter, P, and the word in the next address is placed in the PS register. Finally, the computer enters the FETCH state to initiate the interrupt processing program. The effect of the interrupt is as follows. ADDR is the address supplied by the interrupting device.

```
INTERRUPT state:   comment push down PS;
                   REG[6] ← REG[6]−2;
                   MAR ← REG[6];
                   MEMORY[. . . ;MAR] ← PS;

                   comment push down P;
                   REG[6] ← REG[6]−2;
                   MAR ← REG[6];
                   MEMORY[. . . ;MAR] ← P;

                   comment ADDR is supplied by the device;
                   MAR ← ADDR;
                   MDR ← MEMORY[. . . ;MAR];

                   comment reload P;
                   P ← MDR;

                   comment reload PS;
                   MAR ← ADDR+2;
                   MDR ← MEMORY[. . . ;MAR];
                   PS ← MDR;
                   Change state to FETCH;
```

Each device supplies a unique address to the computer when it causes an interrupt. This address points to the new program counter and processor status for the device processing program. The pair of words in memory at the supplied address are called the *interrupt vector* for the device.

A special instruction, return from interrupt (RTI), is used at the end of the program initiated by the interrupt to restore the status of the machine to that prior to the interrupt. The effect of the RTI is to replace the P register and the PS register by the top element of the stack, pointed to by REG[6], and by the second element of the stack, respectively. Since the PS register is pushed onto the stack before the P register in the INTERRUPT state, the RTI restores them in reverse order and consequently restores the status of the interrupted program. Before issuing the RTI instruction, an interrupt-processing program that uses the stack or otherwise modifies REG[6], the stack pointer, must reset the stack pointer to its value at the time just prior to execution of the first instruction of the interrupt program. Otherwise, the RTI instruction will recall the wrong data into the P and PS registers. If this happens, the link back to the interrupted program is lost.

An interrupt-driven input/output program may be structured as either a coroutine or subroutine, depending on the characteristics of the program. When an interrupt occurs, if the input/output program ought to resume at the point of its previous exit, then it should be a coroutine. If the program is always entered at a unique beginning point, then it can be a subroutine. Since the coroutine structure is the more general structure, we shall focus on coroutines exclusively in the remainder of this discussion.

To see how coroutines can be used for interrupt processing, consider the steps taken by a typical input/output program. When a program suspends itself, we say it "goes to sleep"; and when it is reentered, we say it is "awakened." In these terms a typical sequence of steps for an input/output program is:

1. Initiate the input/output action.
2. Return to the main program, and thereby go to sleep.
3. When awakened by an interrupt, continue from the point of suspension.
4. Perform postprocessing if necessary.
5. Return to step 1 for another iteration if more is to be done; otherwise, exit.

Example 8.4 shows a typical input/output program. The contents of the second word of the interrupt vector (the new PS) are discussed in detail in the next section, on the control of multiple devices.

EXAMPLE 8.4

The following set of instructions illustrates the coroutine linkage of an interrupt-driven input/output program. It shows a coroutine with two distinct segments that initiate an output operation. This is more general and different from the implementation of coroutines with a single output instruction.

```
    LABEL     CODE    OPERAND          COMMENTS
1.  PPB       =       177566     ; Address of the teletype
                                 ; punch buffer.
2.  PUNCH     =       74         ; Address of interrupt vector
                                 ; for the punch.
3.            =       PUNCH
4.                    AWAKE      ; Address of output corou-
                                 ; tine.
5.                    200        ; Processor status word for
                                 ; output coroutine.
    ...               ...           ...
6.  NEXTN:            NEXT       ; The address of one entry
                                 ; point for output program.
7.  NEXTM:            NEXT1      ; The address of another en-
                                 ; try point for output pro-
                                 ; gram.
8.  AWAKE:                       ; Initial entry point for out-
                                 ; put program.
9.            MOV     NEXTN,PUNCH ; In output program, store
                                 ; new return address in inter-
                                 ; rupt vector.
10.           MOVB    OUTDATA,PPB ; Initiate new output opera-
                                 ; tion by placing a word in
                                 ; the punch buffer.
11.           RTI                ; Return to interrupted pro-
                                 ; gram.
12. NEXT:                        ; The entry point when an in-
                                 ; terrupt occurs at the com-
                                 ; pletion of the previous op-
                                 ; eration.
    ...       ...    ...           ...
13.           MOV     NEXTM,PUNCH ; Store another return ad-
                                 ; dress in interrupt vector.
```

```
    LABEL     CODE     OPERAND          COMMENTS
14.           MOVB     OUTDATA,PPB      ; Initiate new output opera-
                                        ; tion.
15.           RTI                       ; Return to interrupted pro-
                                        ; gram.
16.  NEXT1:                             ; The entry point when an in-
                                        ; terrupt occurs at the com-
                                        ; pletion of the previous op-
                                        ; eration.
17.           INC      %1               ; An instruction internal to
                                        ; the output coroutine.
                                          . . .
```

In actuality, Example 8.4 is not quite correct for the following reason. The input/output program cannot disturb the general registers of the computer during the period of time in which the main routine is interrupted unless it restores their original contents before the main routine is reentered. A trivial example of a disruption, such as that implied by Example 8.4, is the destruction of the contents of REG[1] by the input/output program. It is generally not possible to predict whether the main program will be interrupted at a point in which REG[1] contains data to be used subsequently or whether, in fact, the contents of REG[1] can be destroyed. Consequently, if an input/output program makes use of REG[1], it must save the contents of REG[1] upon entry from the main program and restore the contents of REG[1] before it returns to the main program.

We assume that there is a subroutine, called SAVEP, that stores everything that might be disturbed by an input/output program and that a subroutine UNSAVEP restores the saved items to their original contents. Another subroutine SAVEI can be programmed to preserve the state of the input/output program such as REG[0] through REG[5]. Why must REG[6] and REG[7] not be preserved? The general registers REG[0] through REG[5] will be referred to as the *state* of the computer. The subroutine UNSAVEI in this case performs the inverse operation and restores the state of the input/output program. The RTI instructions (which act like coroutine calls) that suspend the input/output program in Example 8.4 ought to be placed in the context shown in Example 8.5 when subroutines for saving registers are used.

Programs for input devices differ from Example 8.5 only in that the MOVB OUTDATA,PPB instruction is replaced by the two instructions MOVB DEVICEBUF,INPUTDATA and INC STATUSREG to initiate the next input operation.

EXAMPLE 8.5

The following program follows the same conventions as Example 8.4, except that it includes the calls on the SAVEP, SAVEI, UNSAVEP, and UNSAVEI subroutines:

```
    LABEL    CODE    OPERAND          COMMENTS
1.  PPB      =       177566     ; Address of the teletype
                                ; punch data buffer.
2.  PUNCH    =       74         ; Address of interrupt vector
                                ; for the punch.
3.  RET      =       %5
4.  .        =       PUNCH
5.                   AWAKE      ; Address of output corou-
                                ; tine.
6.                   200        ; Processor status word for
                                ; output coroutine.
7.  NEXTN:           NEXT       ; The address of one entry
                                ; point for output program.
8.  NEXTM:           NEXT1      ; The address of another en-
                                ; try point for output pro-
                                ; gram.
9.  AWAKE:   JSR     RET,SAVEP  ; Initial entry point for out-
                                ; put program. After entry,
                                ; save state of interrupted
                                ; program.
10.          JSR     RET,UNSAVEI ; Restore the state of this
                                 ; program.
    ...      ...     ...        ...
11.          MOV     NEXTN,PUNCH ; In output program, store
                                 ; new return address in inter-
                                 ; rupt vector.
12.          JSR     RET,SAVEI  ; Save the state of this pro-
                                ; gram.
13.          JSR     RET,UNSAVEP ; Restore state of interrupted
                                 ; program.
14.          MOVB    OUTDATA,PPB ; Initiate new output opera-
                                 ; tion by placing a word in
                                 ; the punch buffer.
15.          RTI                 ; Return to interrupted pro-
                                 ; gram.
```

```
     LABEL     CODE     OPERAND          COMMENTS
16.  NEXT:     JSR      RET,SAVEP        ; The entry point when an
                                         ; interrupt occurs at the com-
                                         ; pletion of the previous op-
                                         ; eration. Save state of inter-
                                         ; rupted program.
17.            JSR      RET,UNSAVEI      ; Restore the state of this
                                         ; program.
     ...       ...      ...                ...
18.            MOV      NEXTM,PUNCH      ; Store another return ad-
                                         ; dress in interrupt vector.
19.            JSR      RET,SAVEI        ; Save the state of this pro-
                                         ; gram.
20.            JSR      RET,UNSAVEP      ; Restore state of interrupted
                                         ; program.
21.            MOVB     OUTDATA,PPB      ; Initiate new output opera-
                                         ; tion.
22.            RTI                       ; Return to interrupted pro-
                                         ; gram.
23.  NEXT1:                               ; The entry point when an in-
                                         ; terrupt occurs at the com-
                                         ; pletion of the previous op-
                                         ; eration.
```

In Example 8.5 the call on the subroutine UNSAVEP occurs before the output operation is initiated, rather than after it is initiated. If the example had been written with the instructions in the order

```
...
MOVB    OUTDATA,PPB     ; Initiate new output operation.
JSR     RET,UNSAVEP
RTI
```

then under certain conditions the program will fail to execute correctly. Specifically, suppose that the output device is so fast that it completes its operation while the UNSAVEP subroutine is in process and before the RTI instruction is executed. Then, when the interrupt is recognized, the output coroutine will be entered and the state of the UNSAVEP subroutine will be saved by the SAVEP subroutine. If the SAVEP subroutine uses the same locations that UNSAVEP uses to restore the state of the original interrupted program, these locations will be overwritten and the state of the original interrupted program will be lost.

From this discussion we see that when an operation for a particular device has been initiated, an interrupt for that device cannot be recognized until after the RTI is executed. The next section describes how to prevent this.

The type of failure discussed here is characteristic of all interrupt programs. The crux of the problem is sensitivity to timing, which greatly increases the difficulty of writing interrupt programs correctly and of analyzing their behavior. An incorrect interrupt program may seem to work properly, but time dependencies can cause it to fail on rare occasions. Interrupt programs require careful design to ensure that all the nuances of timing have been taken into consideration.

8.3 CONTROL OF MULTIPLE DEVICES

The previous section focused on the problem of controlling a single device in an input/output environment. Usually, however, many different input/output devices are in operation simultaneously. In fact, a computer system usually operates at peak efficiency when input/output operations are simultaneous instead of sequential. Suppose, for example, that devices READ and PUNCH operate simultaneously. Examination of Example 8.5 reveals that the subroutines SAVEP, SAVEI, UNSAVEP, and UNSAVEI must be private to the device PUNCH in the sense that they maintain the state for the PUNCH program and must not confuse its state with that of the READ program. Hence the READ program should follow the format of Example 8.5, except that it should use different state-changing routines. It might use routines called SAVEP1, SAVEI1, UNSAVEP1, and UNSAVEI1, for example, to distinguish them from SAVEP, SAVEI, UNSAVEP, and UNSAVEI.

The proliferation of these subroutines can be eliminated, however, and we can use a single SAVE and UNSAVE subroutine together with push-down storage. How many stacks are required in this case? Let us assume that the SAVE and UNSAVE subroutines use the stack pointed to by REG[6] to store and retrieve the state of a program. Now suppose that the main program is executing when an interrupt occurs. After the interrupt and before entering SAVE, the stack is as shown in Figure 8.5.

The SAVE subroutine has to store the state of the main program, and UNSAVE must restore the state of the input/output program. When the interrupt occurs, the input/output program must restore its own state and must have a push-down stack available for its temporary storage. Figure 8.5 shows that the input/output program state is located somewhere in the main program stack, at the top of the input/output stack. After restoring its state, the input/output program has virtually no room on its push-down stack for more temporary storage. The only

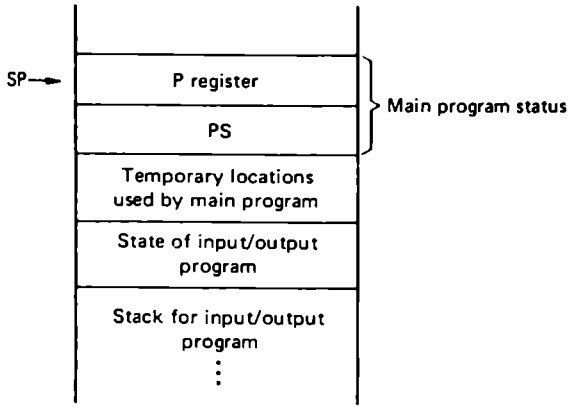


FIGURE 8.5 Contents of machine stack immediately after an interrupt

storage available is that formerly occupied by its state information. The input/output program may not be able to call its own subroutines for lack of push-down storage, and it cannot be suspended if it adds one more word to its push-down stack; for then there is insufficient storage in the push-down stack to hold its state. The problem is that the top of the input/output stack at the point of interrupt is contiguous to the bottom of the main program stack, thus placing a severe limit on the growth of the input/output stack. The solution is to have a stack for each coroutine. In general, coroutines require multiple stacks, one stack for each coroutine.

Assume now that every coroutine has a stack. Different stacks are implemented by changing the contents of the stack pointer, REG[6]. These private stacks are used to hold temporary variables, subroutine linkages, and the state of the coroutine when it is interrupted. Example 8.6 illustrates how one pair of SAVE and UNSAVE subroutines can be used by all the coroutines under these conditions.

EXAMPLE 8.6

The following subroutines save and restore the contents of registers 0 through 5. Note that REG[5] is already on the stack at entry to SAVE owing to the execution of a JSR instruction. Recall SP is REG[6] in the example. The jump (code JMP) instruction replaces the contents of the P register by the specified operand.

```
      LABEL     CODE    OPERAND       COMMENTS
      ...       ...     ...           ...
1.    SAVE:     MOV     %4,-(SP)      ; Save registers on stack.
2.              MOV     %3,-(SP)
                ...
3.              MOV     %0,-(SP)
4.              JMP     @%5           ; Return to calling program by
                                      ; jumping to the location pointed
                                      ; to by REG[5].
                ...     ...           ...
5.              JSR     %5,SAVE       ; A call to SAVE.
      ...       ...     ...           ...
6.    UNSAVE:   ADD     #2,SP         ; Pop the linkage register, REG[5],
                                      ; from the stack.
7.              MOV     (SP)+,%0
8.              MOV     (SP)+,%1
                ...     ...
9.              MOV     (SP)+,%4
10.             RTS     %5            ; The old contents of REG[5] are
                                      ; now on the top of the stack. Re-
                                      ; turn to calling program.
                ...     ...           ...
11.             JSR     %5,UNSAVE     ; A call on UNSAVE.
                ...     ...           ...
```

In the case of the SAVE subroutine, REG[5] is pushed onto the stack by the JSR %5,SAVE instruction and need not be pushed on again. Instead of returning to the calling program with an RTS, SAVE jumps to the location contained in REG[5], which is the return address. To perform this jump, the deferred register addressing mode is required. A JMP instruction is used because the normal branch instructions implicitly assume the relative addressing mode. For UNSAVE, the linkage register is popped from the stack and registers 0 through 4 are restored. At this point the value of REG[5] to be restored is on the top of the stack. Although the contents of the linkage register, REG[5], are lost on entry to UNSAVE, no harm is done because REG[5] normally has been saved by a call on SAVE just prior to the call on UNSAVE. A normal RTS restores REG[5] and returns from UNSAVE.

Now Example 8.5 can be rewritten to control multiple devices. This is done in Example 8.7 by using the SAVE and UNSAVE subroutines of Example 8.6.

EXAMPLE 8.7

The following program follows the same conventions as Example 8.5, except that it utilizes a stack for every coroutine and the common SAVE and UNSAVE subroutines of Example 8.6. Each coroutine has its own private locations OWNSP and SPTEMP for storing its own stack pointer and the stack pointer of the interrupted program, respectively.

```
     LABEL     CODE     OPERAND          COMMENTS
1.   PPB       =        177566           ; Address of the teletype
                                         ; punch data buffer.
     PUNCH     =        74
2.   SP        =        %6
3.   RET       =        %5
4.             =        PUNCH
5.                      AWAKE            ; Address of output corou-
                                         ; tine.
6.                      200
     . . .              . . .              . . .
7.   NEXTN:             NEXT             ; Address of one entry point
                                         ; to output program.
8.   NEXTM:             NEXT1            ; Address of another entry
                                         ; point to output program.
9.   AWAKE:    JSR      RET,SAVE         ; Initial entry point for out-
                                         ; put program. Save state of
                                         ; interrupted program in its
                                         ; stack.
10.            MOV      SP,SPTEMP        ; Save the stack pointer of
                                         ; interrupted program.
11.            MOV      OWNSP,SP         ; Restore stack pointer for
                                         ; this coroutine's stack.
12.            JSR      RET,UNSAVE       ; Restore the state of this
                                         ; program from this stack.
     . . .     . . .    . . .              . . .
13.            MOV      NEXTN,PUNCH      ; Store new return address
                                         ; in interrupt vector.
14.            JSR      RET,SAVE         ; Save state of this program
                                         ; in this stack.
15.            MOV      SP,OWNSP         ; Save the stack pointer of
                                         ; this program.
```

```
    LABEL     CODE    OPERAND          COMMENTS
16.           MOV     SPTEMP,SP        ; Set up stack pointer to
                                       ; interrupted coroutine's
                                       ; stack.
17.           JSR     RET,UNSAVE       ; Restore the state of inter-
                                       ; rupted program from its
                                       ; stack.
18.           MOVB    OUTDATA,PPB      ; Initiate new output opera-
                                       ; tion.
19.           RTI                      ; Return to interrupted pro-
                                       ; gram.
20.  NEXT:    JSR     RET,SAVE         ; The entry point when an
                                       ; interrupt occurs at the
                                       ; completion of the previous
                                       ; operation. Save state of
                                       ; interrupted program in its
                                       ; stack.
21.           MOV     SP,SPTEMP        ; Save the stack pointer of
                                       ; interrupted program.
22.           MOV     OWNSP,SP         ; Restore stack pointer for
                                       ; this coroutine's stack.
23.           JSR     RET,UNSAVE       ; Restore the state of this
                                       ; program from this stack.
      . . .             . . .          . . .
24.  SP:                STACK+2        ; Stack pointer for this stack.
25.  SPTEMP:            0              ; Stack pointer for inter-
                                       ; rupted program's stack.
26.  STACK:.    =       .+200          ; Reserve space for 200-byte
                                       ; stack.
```

When two or more input/output devices are active simultaneously, what happens if more than one interrupt occurs at the same time? Clearly, the computer cannot respond to more than one interrupt at a time. Therefore, interrupts are usually ordered by a priority-weighting scheme.

The priority of a program and of an interrupting device is determined by an integer called its *priority level.* The PDP-11 has eight priority levels, numbered 0 through 7. In the PDP-11 computer each of the peripheral devices is assigned one of the four priority levels (4 through 7), which cannot be changed by a program. The program currently in execution has its priority level determined by the setting of bits 7 through 5 in the PS register as shown in Figure 8.4. For greater flexibility, the interrupt-enable bits in the device status register can be used to dis-

able interrupts for the corresponding devices even when they otherwise have priority. An interrupt for a device with its interrupt-enable bit reset is simply held in abeyance until the bit is set.

Priority 7 is the highest priority level. The PDP-11 computer recognizes the interrupt for the device with the highest priority with an interrupt-enable bit set, provided the priority is greater than that of the currently executing program. If an interrupt occurs for a device that is not recognized immediately because another device has priority, then that interrupt is held in abeyance. It can be recognized later provided the interrupt-enable bit is set to 1, the device has priority over all other devices with interrupts pending, and its priority is greater than that of bits 7 through 5 in the PS register. In the case of two devices that simultaneously satisfy all these criteria, conflicts are resolved by the relative physical position of the cables connecting the devices to the computer. If a high-priority interrupt of an enabled device occurs during the processing of a lower-priority interrupt, it is recognized immediately, and control transfers to the appropriate interrupt location.

When two or more devices are operating simultaneously under interrupt control, interrupts for one device can occur while the program for the other device is in execution. If the input/output programs use the conventions shown in Example 8.7, then they can operate without interactions that lead to failure. In this way a high-priority device can interrupt the program for a low-priority device without failing, and a low-priority interrupt can be held in abeyance while a program for a high-priority device is in execution.

To illustrate, suppose a low-priority device is being processed when a high-priority interrupt occurs. The interrupt generates a return address, saves the status of the low-priority program, and enters the program for the high-priority device. The first action of the high-priority program is to preserve the state of the low-priority program. When high-priority processing is complete, the state and status of the low-priority program are restored, and control returns to the low-priority program at the point of interruption. Thus the high-priority interrupt processing causes a delay between some pair of instructions in the low-priority program. The only possible timing problem arises when such a time delay occurs after a low-priority operation is initiated and before the low-priority program reaches its exit. In this case the low-priority input/output operation may be completed before the low-priority-program exit is taken. If the low-priority program follows the conventions of Example 8.7, the only possible problem is if the delay occurs between the MOVB and RTI instructions.

To see what happens when this delay occurs, assume that a high-priority device interrupts a low-priority device between the MOVB and

RTI instructions. Further, assume that the interrupt processing program has the same priority as the interrupting device. (Most interrupt processing programs are written in this manner.) If the low-priority device completes its operation, then it posts an interrupt, which is not recognized because the interrupt does not have higher priority than the program in execution. As higher-priority processing finishes, control returns to the RTI instruction, which finds everything in place just as if no high-priority processing had occurred. The RTI instruction is executed, returning control to a program of still lower priority. At this point the interrupt for the low-priority device is recognized, and the low-priority routine is reentered at the correct reentry point.

We have already seen that Example 8.5 (and hence Example 8.7) is logically correct for arbitrary delays between other instructions. Therefore, a delay between an MOVB and an RTI instruction costs a finite amount of time and some memory for holding status and state. At any given point in time at most four programs can be suspended owing to interrupt processing, since there are only four priority levels for peripheral devices. Hence, Example 8.7 is correct provided push-down stack memory is sufficiently large to service four high-priority interrupts.

Now suppose that a low-priority interrupt occurs during the processing of a high-priority device. By assumption, the priority in the PS register is set when the high-priority program is entered and does not change until an RTI is executed. Hence the low-priority interrupt is held in abeyance until the high-priority program initiates a new high-priority operation and executes an RTI. As a result, the effect is the same as if the low-priority interrupt had occurred immediately after completion of the high-priority processing. Thus one input/output program does not interrupt another in this case, and no timing failures are introduced.

The program that services an interrupt need not be at the same priority level as the device causing an interrupt. By changing the PS register in the interrupt vector, any priority can be selected for an interrupt program. Hence an interrupt program can respond to interrupts from a selected set of devices, namely, those whose priority is greater than that assigned to the program. It might seem that, if an interrupt program is assigned a priority lower than that of an interrupting device serviced by the program, the device can continually interrupt its own service program without ever receiving service. Fortunately, the PDP-11 is constructed to avoid this situation. An interrupting device cannot initiate another interrupt until its buffer register is altered.

However, the program can still fail in a very subtle manner. Reconsider Example 8.7 and assume the interrupt processing program has a priority lower than that of the device it services.

If we use a hypothetical ultrafast output device, it might generate

an interrupt immediately after the MOVB instruction initiates a new operation and before the RTI instruction is executed. If this occurs repeatedly, successive copies of the PS and P registers are pushed onto the stack until the stack finally overflows. The device need not be a hypothetical device, since we have seen that interrupts can cause an arbitrary delay between any two instructions of a program. Hence, for Example 8.7 to be correct in all cases, we force the interrupt processing program to have the same (or higher) priority level as the interrupting device. Otherwise, Example 8.7 can fail as outlined above when a device can interrupt its own service routine.

Although the conventions in Example 8.7 are completely general, some programmers prefer to disable the interrupt system while an input/output program is executing and enable it after the RTI. An interrupt that occurs while the interrupt system is disabled is thus held in abeyance until the interrupt system is enabled again. This can be accomplished by setting the input/output program priority to 7. Hence, while an input/output program is executing, no interrupts are honored.

However, this method is less desirable when high-priority devices should truly be given priority over low-priority devices. When the interrupt system is disabled, high-priority processing waits until low-priority processing has terminated. When the interrupt system remains enabled during input/output processing, low-priority processing is interrupted in favor of high-priority processing.

8.4 DIRECT MEMORY ACCESS CHANNELS

The input/output described in the previous sections is performed through the execution of a program. Thus during input/output operations the central processor is in the data flow path between main memory and the outside world.

System/360-370 computers and certain devices on the PDP-11, as well as many other computers, use special input/output mechanisms to remove the central processor from the input/output data flow path. This increases the potential computation speed of the computer system by decreasing the load on the central processor. The input/output mechanism makes use of a special input/output processor called a *data channel* or *direct-memory-access channel*, which can access main memory independently of the central processor.

A typical computer system with two data channels is shown in Figure 8.6. Note that three different data flows can be supported in the computer system: one between the central processor and memory and one through each data channel between memory and peripheral devices. Because the memory shown in Figure 8.6 has only one MDR and MAR

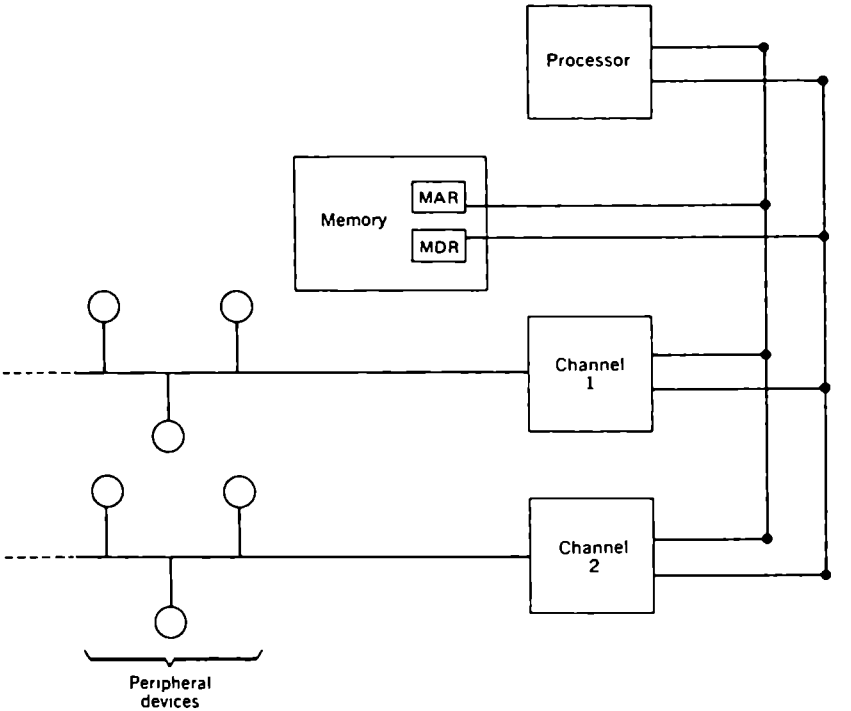


FIGURE 8.6 A computer system with two data channels for input/output operations

register pair, only one datum can be fetched or stored at a time. Hence conflicts can arise when two independent requests reach memory simultaneously.

There are several ways of resolving memory access conflicts. A straightforward method is to establish priorities for access and to defer a low-priority memory access in favor of high-priority memory access. Usually the data channels are given priority over the central processor because the channels operate in a mode in which timing is critical, whereas the central processor can be delayed without ill effect. In any event, this method essentially time-shares the memory by interleaving the memory requests of the data channels and the central processor. As a result the computational speed of the central processor tends to decrease when input/output activity increases, but the decrease is much less than that in a system without data channels.

High-performance computers are often organized with modular memories in order to reduce the performance degradation due to memory access conflicts. Figure 8.7 shows a computer system in which main memory consists of four memory modules, each with its own MDR and MAR register. Each word in this memory has a unique address, just as in the

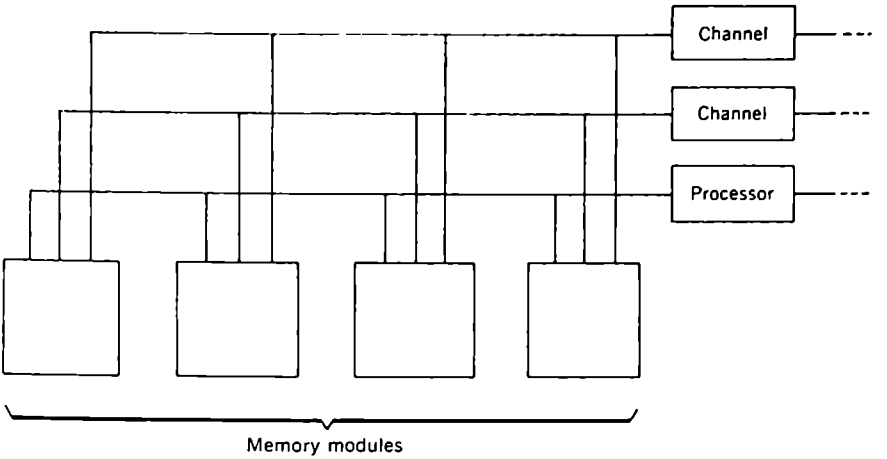


FIGURE 8.7 A modular memory system. Each line represents both the address and the data lines shown in Figure 8.6

memory shown in Figure 8.6, and each memory module contains one-fourth of the memory addresses. The advantage of the modular memory is that all modules can honor memory requests simultaneously. Thus if the data channels and the processor request data from different modules simultaneously, no memory conflict occurs. Memory conflicts can still occur in this computer system, so that the processor performance can degrade during periods of high input/output activity. However, conflicts will tend to be much less frequent than in the system shown in Figure 8.6, and therefore processor performance is much less sensitive to input/output activity.

Since data channels are essentially processors, in many computers channels are controlled by programs analogous to those that control the actions of a central processor. The programs may be fixed in the computer hardware or stored in memory and executed much like ordinary programs. The PDP-11 uses the former approach and the System/360-370 the latter.

A direct-memory-access channel (DMA channel) for the PDP-11 is normally an integral part of the peripheral device it serves. A PDP-11 DMA channel has four registers: a command-and-status register, an address register, a data register, and a word-count register. The processor initializes the address register with the address of an area in memory into which data are to be stored (input) or from which they are to be retrieved (output). The word-count register is initialized with the number of words that are to be transferred. Then the processor initiates the device by storing a command in the command-and-status register.

If the DMA device is an input device, it places a datum in the data

register and requests access to the memory. The request is granted for the next memory cycle. The address and data are transferred to the MAR and MDR, respectively, and the memory is instructed to perform a write operation. After the one memory cycle the DMA channel relinquishes control of memory, decrements the word-count register, increments the address register, retrieves another datum, and repeats the process. During this time the processor and other channels have access to memory. When the word-count register reaches zero, the channel issues an interrupt to the processor to inform it that the block of data has been transferred. At this point the processor can reinitialize the device or leave it idle.

The System/360-370 data channel actually executes a stored program. The major difference between channel programs and ordinary programs is that data channels do not perform the same operations as a central processor, and they therefore require a different instruction set. Some of the System/360-370 channel commands are listed in Table 8.2. Note that each channel command causes an action very similar to an action of a PDP-11 DMA channel. Several of these actions can be linked into a program. Another major difference between the PDP-11 DMA channel and the System/360-370 channel is that the former usually controls only one device and the latter may control several different devices.

TABLE 8.2

CHANNEL COMMAND	PARAMETERS	ACTION
READ	BUFFER,N	Read N bytes into successive bytes of memory, starting at address BUFFER.
WRITE	BUFFER,N	Write N bytes from successive bytes of memory, starting at address BUFFER.
TRANSFER IN CHANNEL	NEXTCOMMAND	Take next channel command from address NEXTCOMMAND.

Figure 8.8 shows how the control information is stored in a channel command word. The shaded areas hold data whose function is very specialized and is not discussed in this text.

In normal operation an input/output program constructs a sequence of channel commands and then issues an instruction that causes a channel to execute the channel program. Each channel command has a bit called a *chain command bit*, which is used like a HALT instruction. If the chain command bit of a channel command is 0, then the channel termi-

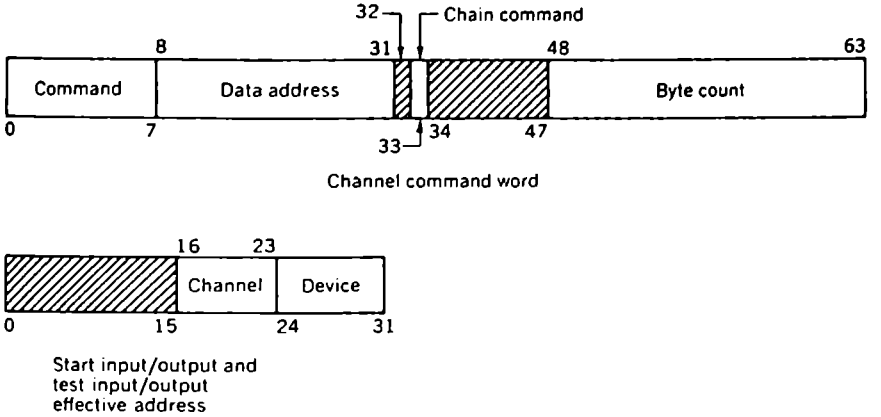


FIGURE 8.8 System/360-370 input/output control-word representation

nates operation when that channel command has been executed. If the chain command bit is 1, the channel executes the next command in the channel program. When a channel operation terminates, the end of operation is signaled by an interrupt. Thus input/output operations with data channels can be controlled by interrupt programs similar to those discussed in the previous section.

Since the channel commands above do not specify which device or channel is involved in an input/output operation, this information is specified in the instruction that initiates a channel operation. Hence the System/360-370 input/output instructions include the following:

INSTRUCTION	CODE	PARAMETERS	ACTION
START I/O	SIO	CHANNEL,DEVICE	Initiate an input/output operation on specified channel and device.
TEST I/O	TIO	CHANNEL,DEVICE	Set condition code to indicate present status of channel and device.

The base and displacement fields of these instructions encode the channel and device. Execution begins with the computation of the effective address by adding the contents of a base register to a displacement. Then the effective address is interpreted as a channel number and device number, as shown in Figure 8.8.

When the START I/O instruction is issued, the specified channel begins executing a channel program. By convention the address of the first channel command is stored in memory address 72. Execution of START I/O also sets the condition code to indicate whether the operation has

been successfully initiated. The condition code setting of 0 denotes the normal case, and other settings indicate unusual conditions such as a busy channel, which require special processing. The TEST I/O instruction is equivalent to the PDP-11 instruction that tests the state of the READY/DONE bit. The action of the TEST I/O instruction is to set the condition code to indicate the state of the channel and device. In Example 8.8 the condition code setting of 0 indicates that the channel and device are inactive.

Example 8.8 is intended as a sketch of techniques for performing data-channel input/output. Many details have been omitted by necessity. The System/360-370 interrupt system has not been discussed here, but, of course, it is of central importance for input/output programming. This interrupt system functions much like the PDP-11 interrupt system, but it differs considerably in its details. When an interrupt occurs at the completion of an operation, the computer branches to a location in memory that is reserved for that interrupt. As in the PDP-11 program, a return address is generated and stored, and then control passes to an input/output program.

EXAMPLE 8.8

The following program for System/360-370 reads a punched card with 80 characters into memory. The program uses a data channel for input/output and a wait loop to test for the end of an operation. We assume that interrupts cannot occur. The store multiple instruction (code STM) causes all registers in the range indicated to be stored starting at the given memory address.

	LABEL	CODE	OPERAND	COMMENTS
	...	...	...	...
1.	CCW	DS	D'0'	Reserve double word for a channel command word.
2.	CCWN	DC	A(CCW)	Address of channel command word.
3.	READCMD	DC	F'2'	Code for a READ operation in channel command word.
4.	BUFFN	DC	A(BUFF)	Address of buffer storage area.

	LABEL	CODE	OPERAND	COMMENTS
5.	BUFF	DS	80C'	Reserve 80 bytes for storing data on one punched card.
6.	COUNT	DC	F'80'	Number of characters to be read.
7.	CHANNEL	DC	F'1'	The channel number.
8.	READER	DC	F'3'	Device number for card reader.
9.	TEMP	EQU	2	A temporary register, the even register of a double register.
	...	...	...	...
10.		L	TEMP,CCWN	Move address of channel command word to location 72.
11.		ST	TEMP,72	
12.		L	TEMP,READCMD	Begin construction of channel command word. Load code for READ operation.
13.		SLL	TEMP,24	Align this command into leftmost byte.
14.		A	TEMP,BUFFN	Add address of buffer.
15.		L	TEMP+1,COUNT	Load byte count into right end of double register. The chain command bit is 0.
16.		STM	TEMP,TEMP+1,CCW	Store REG[TEMP;TEMP+1] as a channel command word.
17.		L	TEMP,CHANNEL	Construct channel and device information for SIO instruction. Load channel number.
18.		SLL	TEMP,8	Shift left one byte.
19.		A	TEMP,READER	Add device number.
20.		SIO	0(TEMP)	Start operation.
21.		BC	1+2+4,ERROR	Branch to ERROR on any condition code except 0.
22.	TEST	TIO	0(TEMP)	Test for input/output completion.
23.		BC	1+2+4,TEST	Loop until operation is done.
24.		L		At this point 80 bytes have been read into the buffer.

Very few programmers ever write input/output instructions for System/360-370 computers. Instead, these computers normally operate under the control of an executive program. One of the primary functions of the executive program is to do all the input/output processing and thus relieve the ordinary programmer of the problems related to input/output programs. To perform input/output, the user programs have to call standard input/output subroutines that are part of the executive program. System/360-370 designers were particularly careful to prevent ordinary users from executing input/output instructions directly. When a user program is in process, the input/output instructions are not executable, and any attempt to execute such an instruction causes an interrupt that branches to an error program.

8.5 ASYNCHRONOUS CONTROL OF INPUT/OUTPUT

We have seen that input/output operations in the PDP-11 and System/360-370 computers can be executed concurrently with computation in the processor. Most peripheral devices operate at data transmission rates much lower than those a computer can sustain, because peripheral devices normally have mechanical motion of some kind, whereas computers do not. During input/output operations the computer is constrained to transmit information at a rate sustainable by the peripheral devices, which can result in a significant decrease in computation rate. For maximum efficiency during an input/output action we wish to compute at the fastest possible rate within the constraints imposed by the peripheral device, and we can do this only if the computation and input/output actions are correctly synchronized.

Consider, for example, the following outline, which is typical of many programs:

1. Read a set of input data.
2. Compute with these data, producing a collection of results.
3. Output the results.
4. If there are more input data, return to step 1; otherwise halt.

The least efficient use of the computer for this program is illustrated in Figure 8.9*a*, where we see that there are no concurrent operations. In this example processing is suspended immediately after the computer issues an input or output operation, and processing resumes when the corresponding REPLY signal is received. The behavior here is typical of programs that use wait loops for input/output operations.

In Figure 8.9*b* we see an example of an interrupt-driven program

that is somewhat more efficient than the previous example. Here computation continues after input/output operations have been initiated. The periods of no computation occur because the input device is a bottleneck. When computation reaches a point at which new input data are required, processing is suspended until the input device completes its operation. The output device also is suspended for periods of time while waiting for the computation to generate new output data. Immediately after each input operation terminates, a new input operation is initiated, so that the input device runs continuously.

Figure 8.9*c* is similar to Figure 8.9*b* except that the bottleneck is the output program instead of the input program. Now computation must be suspended while the output device performs an operation, and the input device, in turn, is suspended periodically until program processing reaches a point at which it can accept new data. As in the previous example, input/output operations can proceed concurrently with computation, but the effective computation rate is determined by the speed of the output device.

We wish to write our programs so that they automatically adopt the behavior shown in Figure 8.9*b* and *c* depending on which device is slower. That is, the slowest device should be made to run continuously by initiating a new operation on that device immediately after an operation terminates. Moreover, concurrent operation of input/output devices and processor should occur as much as possible. Automatic suspension of activity should occur whenever one part of the system has to wait for another. The behavior we desire should also be able to adapt to complex situations such as the one shown in Figure 8.9*d*. Here we see a case in which the input device is a system bottleneck periodically and should run continuously at such times. At other times the output device is the bottleneck, so it should be controlled to run continuously. In either case, the processor should perform computation until it reaches a point at which it must suspend activity pending completion of an input or output operation.

To perform input/output operations with high efficiency, we construct the input/output programs as coroutines and use the interrupt system to generate calls on these coroutines from the main program. We avoid the inefficient behavior shown in Figure 8.9*a* by allowing the input/output coroutines to operate slightly out of step with the main program. Thus an input coroutine might work ahead of the main program by initiating new input operations during periods when the input device might otherwise be idle and before the input data have been requested by the main program. Conversely, the main program might work ahead of a slower output coroutine instead of waiting in order to remain in step.

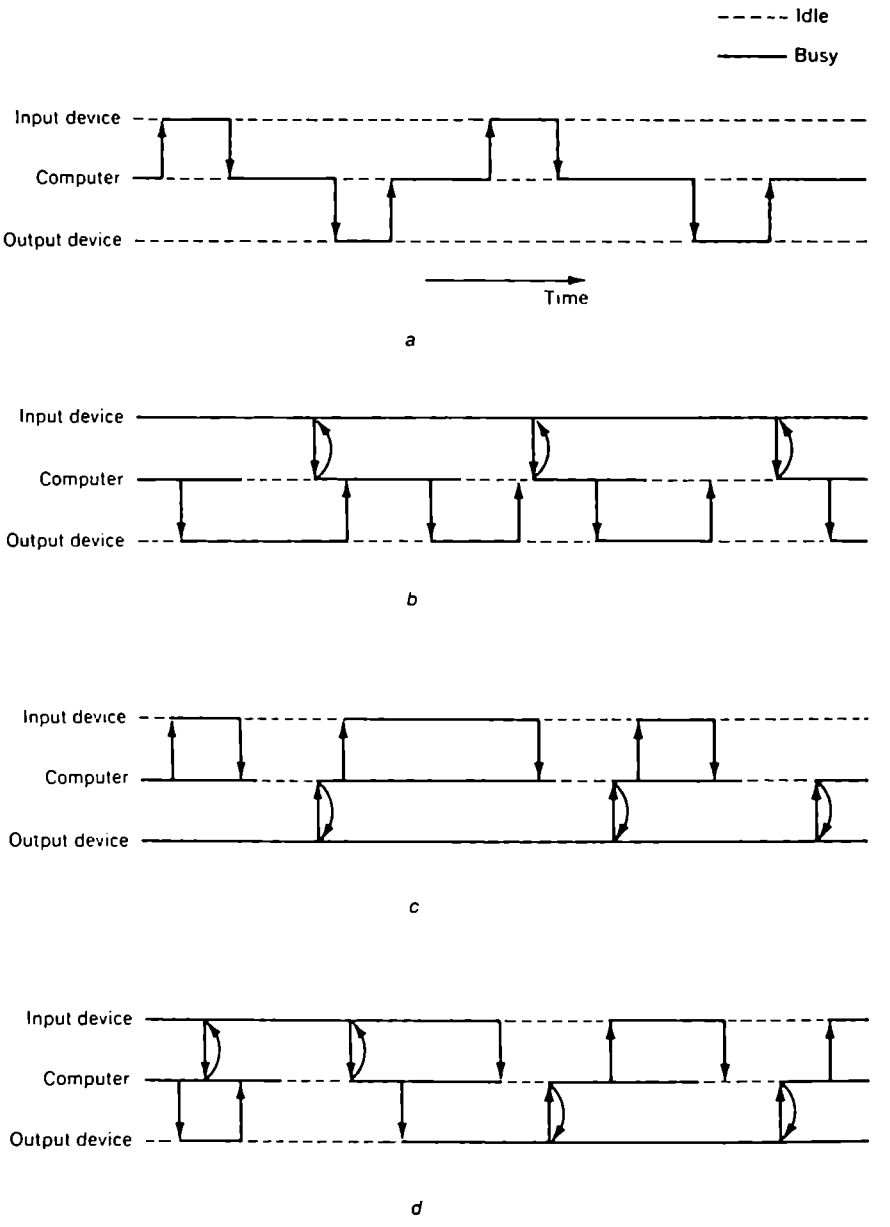


FIGURE 8.9 A study of the activity of a typical program. The arrows that leave a line marked COMPUTER are input/output requests, and those that enter are replies. (*a*) No interrupts; (*b*) with interrupts, computation limited by input; (*c*) computation limited by output; (*d*) computation limited by both input and output

To produce this behavior, all data are exchanged between the input/output coroutines through queues, rather than by the usual methods for passing parameters. Figure 8.10 shows how queues are placed between the input/output coroutines and the main program. Consider, for example, a short time interval in which the input device transmits data faster than the main program can accept data. The input coroutine places the first item in the queue and immediately initiates a new input operation. When this operation is complete, the new data are queued, and another operation is initiated. This process is repeated as each input operation is completed. The queueing process places the data where they can be retrieved by the main program in the order in which they were processed by the input coroutine. The advantage of the queue is that for brief periods the main program can process data at a rate different from that at which they are processed by the input coroutine.

The input coroutine reads information from a peripheral device and places it in the input queue. Similarly, the output coroutine picks up information from the output queue and transmits it to the output device. The input and output queues permit us to do computation concurrently with input/output, as shown in Figure 8.9*b* and *c*, instead of as shown in *a*. Moreover, the queues can smooth variations in input or output data rates of the type shown in *d*. In *b* the computation speed is determined by the input device, and the input queue is empty in the steady state. Whenever the input coroutine places new information in the input queue, the main program removes it immediately. In *c* the output queue is normally full. In *d* neither queue may be empty or full in the steady state. For this case the queue tends to permit computation at a rate limited by the average input/output rate rather than by the instantaneous minimum of input and output rates.

The size of the queue has no effect on the computation rate in Figure 8.9*b* and *c*, provided that the queue contains at least two elements. When variations in input/output rates can occur, as in *d*, the computation rate will tend to increase as the queues get larger, until the queues

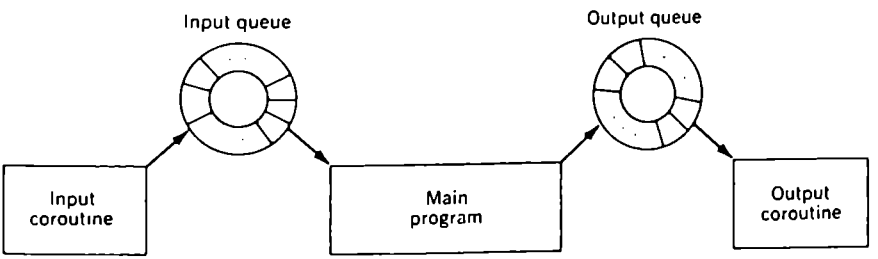


FIGURE 8.10 A program that uses input/output coroutines and queues

are large enough to smooth out the peak variations. When queue sizes are increased beyond this point, the computation rate will not increase.

Although Figures 8.9 and 8.10 illustrate the principles of programming with queues, there are significant details that warrant explanation. The main queue operation is rather straightforward. For example, the input coroutine places new data at the tail of the input queue, and the main program removes data from the head of the input queue. Output coroutine activity is analogous. Figure 8.11 shows a flow chart of the basic behavior of an input and output coroutine operating with a main program through input and output queues. The input coroutine is entered when an interrupt occurs at the completion of an input operation. The coroutine enters the new datum into the input queue, initiates a new input operation, and then returns from the interrupt. The new input

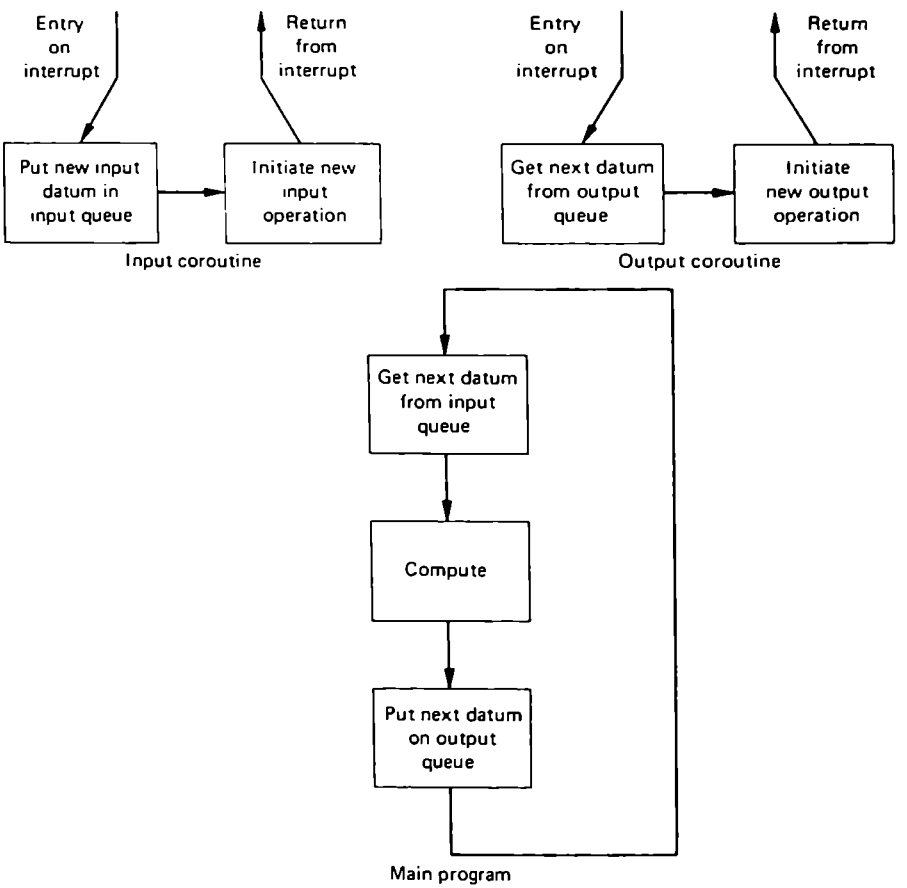


FIGURE 8.11 Flow charts of a typical main program and input and output coroutines

operation eventually will terminate and issue an interrupt to restart the input coroutine. Thus the input coroutine is in an infinite loop as shown. In actual situations, the infinite loop is terminated when the input coroutine detects certain end-of-input conditions. We also do not show how the input coroutine is initialized when processing begins.

Note in Figure 8.11 that the output coroutine behaves in essentially the same way as the input coroutine, except that the data flow is from the main program to the outside world rather than the other way around. The main program in this example is a typical program that repeats the sequence of input, compute, and output until input data are exhausted. We have specifically left out details concerning program initialization and termination to focus on the essentials of the program.

If we could guarantee that queues do not become empty or full, then the flow chart of Figure 8.11 would be sufficient for the simultaneous input, compute, and output operations. However, queues can fill or empty unpredictably during program execution, so that we must treat these cases explicitly to have a correct program.

Figure 8.12 is an embellished flow chart that shows how to deal with full or empty queues. We have added two types of processes, one a WAIT and the other a SIGNAL. A coroutine cannot pass a WAIT until the corresponding condition is satisfied. If the condition is not satisfied, the coroutine is suspended. Thus the input coroutine waits until the input queue is not full before entering a new item. The fact that the queue is not full is reported by the SIGNAL issued by the main program when it removes an item from the input queue. A suspended coroutine is awakened by a corresponding SIGNAL from another coroutine. In this flow chart each WAIT for a condition has a corresponding SIGNAL issued by another coroutine when the condition is satisfied.

Now let us examine the details of the implementation of the WAIT and SIGNAL operations. In this discussion we shall investigate one possible implementation of these processes that works correctly for a large class of computers. In many implementations, WAIT and SIGNAL are functions of an executive program, which often has privileged instructions at its disposal. The privileged instructions enable the executive program to do the WAIT and SIGNAL, and since they are not available to user programs, only the executive program can perform these operations. Here we assume that there is no executive program and that any program can execute the instructions necessary for WAIT and SIGNAL.

First we focus on the suspension of the input coroutine when the input queue is full and show how to restart this coroutine when one or more items are removed from the queue. Details of what happens in this case are shown in Figure 8.13. When the input queue is full, the input coroutine suspends itself by returning to the point of interruption, but

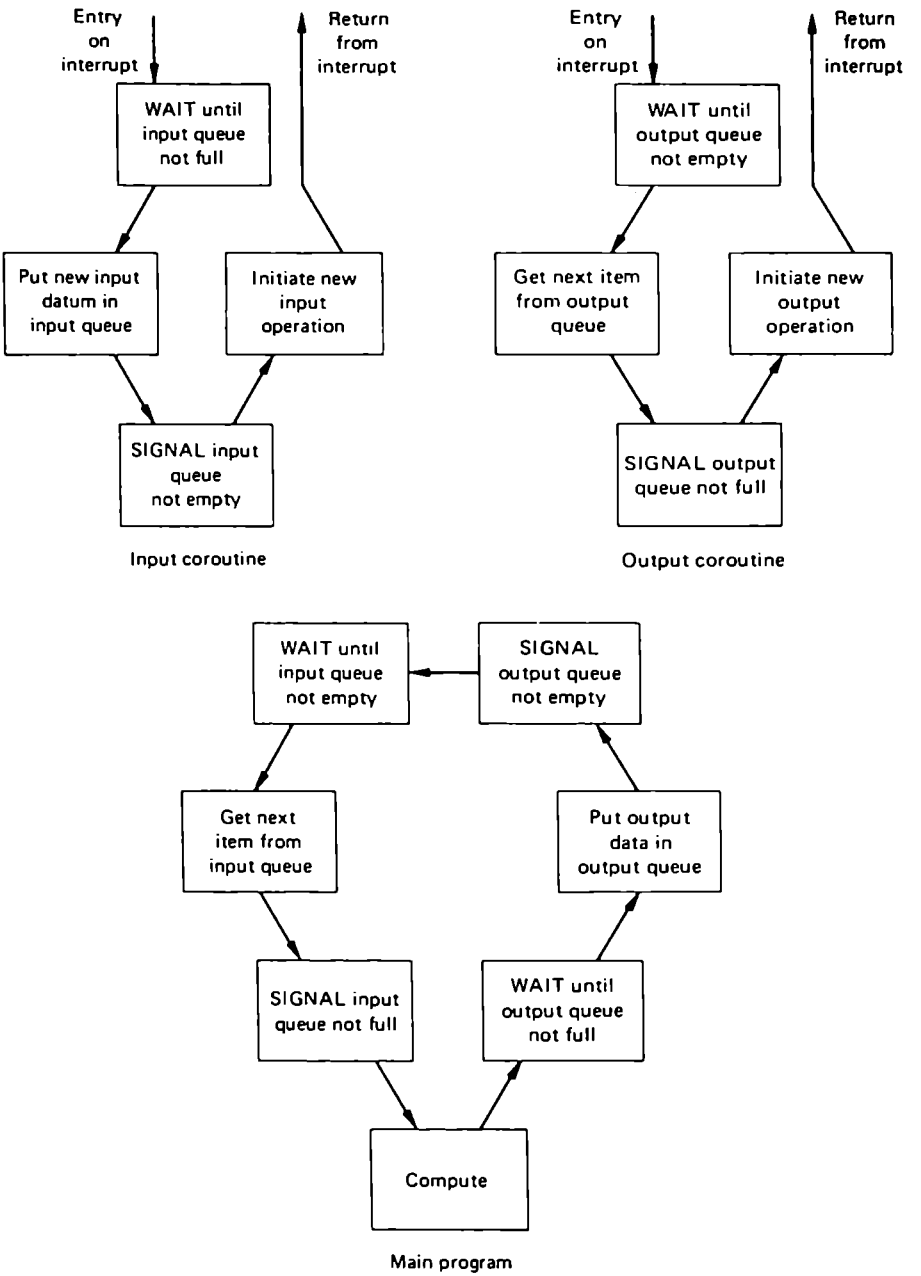

Entry on interrupt
Return from interrupt
WAIT until input queue not full
Put new input datum in input queue
Initiate new input operation
SIGNAL input queue not empty
Input coroutine
Entry on interrupt
Return from interrupt
WAIT until output queue not empty
Get next item from output queue
Initiate new output operation
SIGNAL output queue not full
Output coroutine
WAIT until input queue not empty
SIGNAL output queue not empty
Get next item from input queue
Put output data in output queue
SIGNAL input queue not full
WAIT until output queue not full
Compute
Main program

FIGURE 8.12 Coroutine cooperation with WAITs and SIGNALs

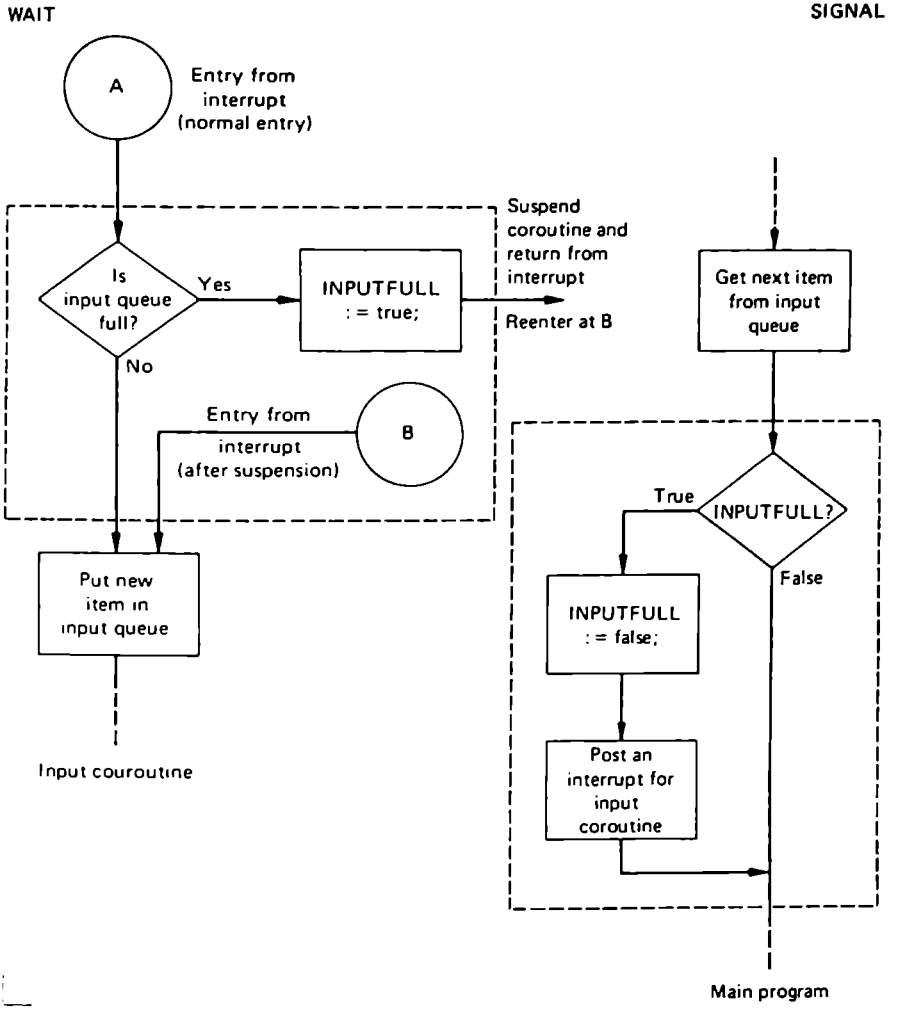


FIGURE 8.13 The implementation of WAIT and SIGNAL

without initiating a new input operation. It must return, in this case to the main program, to allow computation to continue to the point at which an item is removed from the input queue. In Figure 8.13, the suspension of the input coroutine is recorded in the variable INPUTFULL. When the main program removes an item from the input queue, it tests this variable to determine if the input coroutine is suspended while waiting for room in the input queue. If so, the main program reawakens the input coroutine by posting an interrupt, just as if an input device had completed an operation. The input coroutine regains control when the interrupt is recognized, and it restarts from the point of program suspension. This is the entry point labeled B in Figure 8.13. Some com-

puters have instructions that cause interrupts that are indistinguishable from interrupts from peripheral devices. Later in this chapter we shall see how to simulate an interrupt on the PDP-11 computer without such special instructions.

The portions of a flow chart inside dashed boxes in Figure 8.13 implement the WAIT and SIGNAL operations shown in Figure 8.12, except for WAITs issued by the main program. The main program in this example cannot suspend itself as an input/output coroutine can, because it cannot pass control to another program. It must wait by entering a wait loop. In general, in any computation with input/output operations there must exist at least one wait loop in the program in case computation is much faster than input/output. Most modern computer systems have the WAIT and SIGNAL functions implemented as part of the READ and WRITE operations, which are executive functions. A main program calls for input/output through these executive functions. If suspension of the main program is necessary, then the suspension is done by delaying return from READ and WRITE, and thus the wait loop is embedded in the executive program functions and need not be written explicitly in the main program.

For the case in which the main program must suspend itself, one possible implementation is shown in Figure 8.14. Here the main program

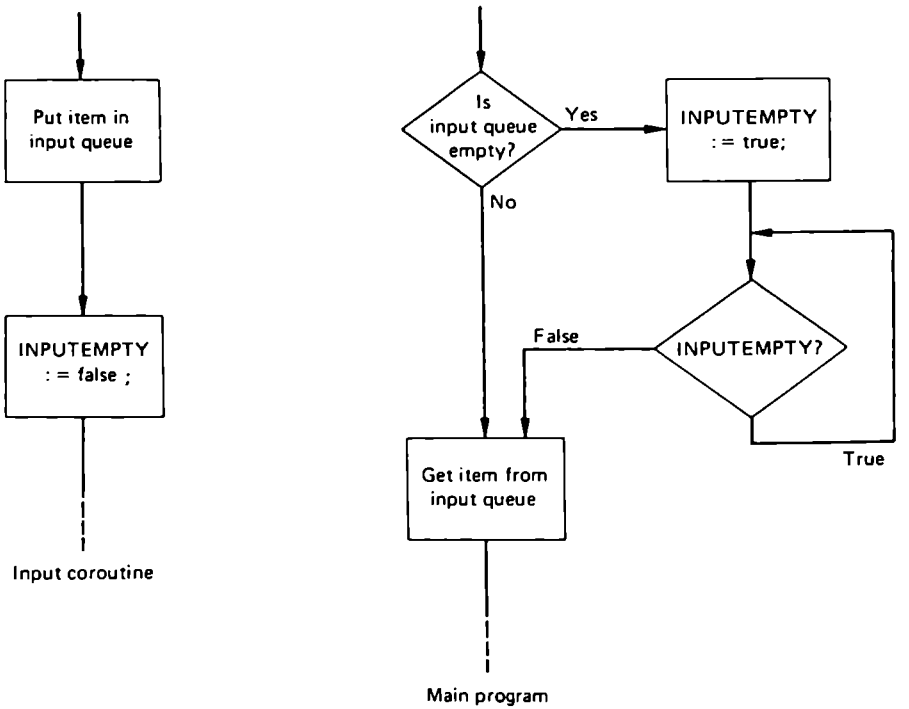


FIGURE 8.14 The suspension of a main program

examines the input queue to determine if it is empty; if so, it sets the variable INPUTEMPTY true and then enters a loop. The input coroutine merely sets INPUTEMPTY false and continues. Eventually, control is returned to the main program, which then leaves the loop in Figure 8.14 because INPUTEMPTY has changed value. This implementation of the WAIT and SIGNAL differs from the implementation of Figure 8.13 only because control is guaranteed to return in the main program in Figure 8.14, whereas the input coroutine in Figure 8.13 must be awakened by an interrupt. Otherwise, control will not return to the input coroutine.

In previous examples of interrupt-driven input/output programs the interrupt system was always on. However, in this case it is absolutely necessary to turn the interrupt system off while manipulating the queues and inspecting their pointers. We have a situation in which two different programs inspect the HEAD and TAIL pointers of the queue and take actions conditioned on their values.

To see that a program failure can occur if interrupts are not turned off, consider the following situation.

1. The main program output section inspects the queue pointers and finds that the queue is full.
2. The output coroutine interrupts the main program output section.
3. The output coroutine inspects the queue pointers and finds that the queue is full.
4. The output coroutine removes one item from the queue.
5. The output coroutine checks to see if the main program output section is suspended and finds that the output section is active.
6. The output coroutine initiates a new output operation and returns.
7. The return passes control back to the main program output section.
8. Before the main program output section can proceed, an output interrupt is recognized and control passes to the output coroutine. (This is quite possible if many different devices are active simultaneously and interrupts from these devices prevent the main program output section from continuing execution.)
9. The output coroutine repeats steps 3 through 6, possibly several times until it empties the queue.

10. Control returns to the main program output section, which, having found the queue full, marks itself suspended and exits.
11. An output interrupt reactivates the output coroutine.
12. The output coroutine inspects the queue pointers and finds that the queue is empty.
13. The output coroutine suspends itself.

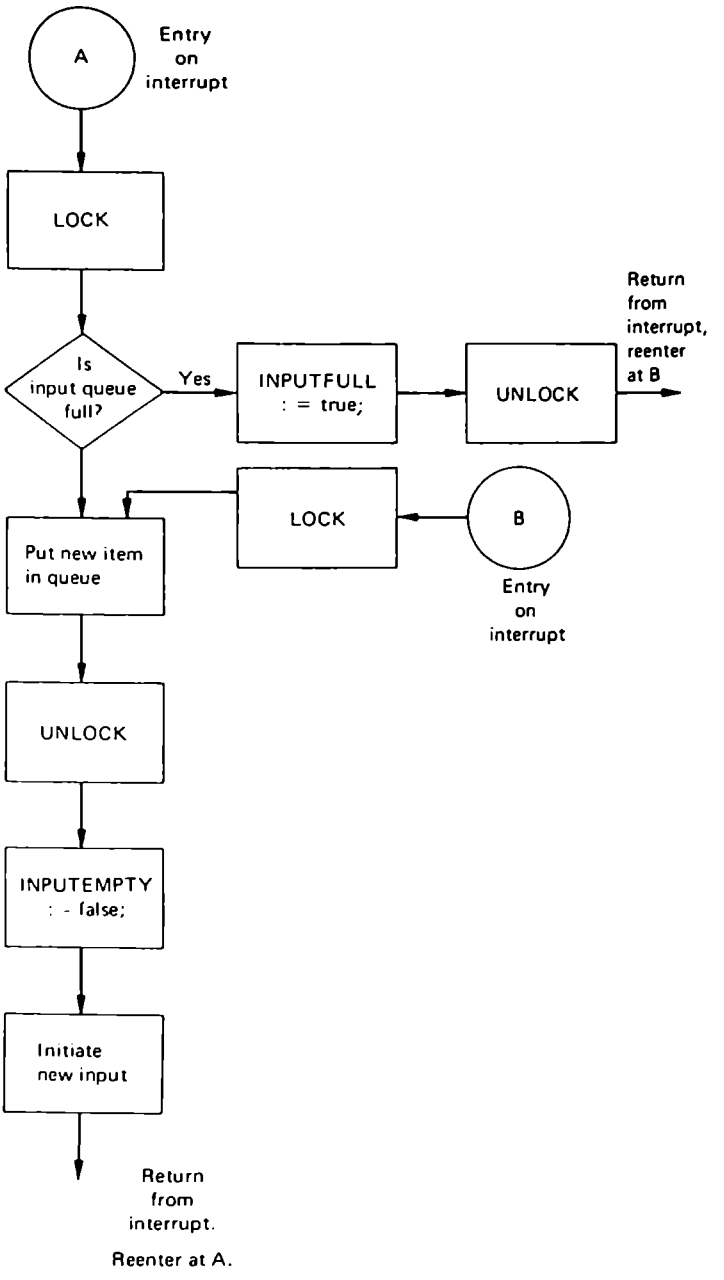
When the last condition arises, both the main program output section and the output coroutines are suspended, and neither can be awakened. The program has failed. A similar failure can occur between the input coroutine and the main program input section.

In the situation above, both the main program output section and the output coroutines observed particular data (the queue pointers) and took actions based on the values of the pointers. The failure occurred because the values of the pointers can change at unexpected times in an interrupt environment. Once again we see that subtle timing problems can lead to unpredictable behavior of input/output programs.

The points in the main program and input/output coroutines at which the interrupts should be turned on and off are indicated in Figure 8.15. The LOCK statement turns off the interrupt system and thereby locks out interrupts. The UNLOCK statement is the inverse operation. Interrupts that occur while the system is off are not lost; they are deferred until the interrupt system is turned on again. Obviously, an entire input/output coroutine can be LOCKed, that is, run with the interrupt system off, but this is unnecessarily conservative and can lower potential performance. The LOCK and UNLOCK actions shown in Figure 8.15 have been inserted so that the queue subroutines and the WAIT processes are uninterruptable. Failures of the type noted above occur because one coroutine can inspect a queue while performing a WAIT and thereby interfere with another coroutine that is altering the state of the queue by inserting or removing an item from the queue.

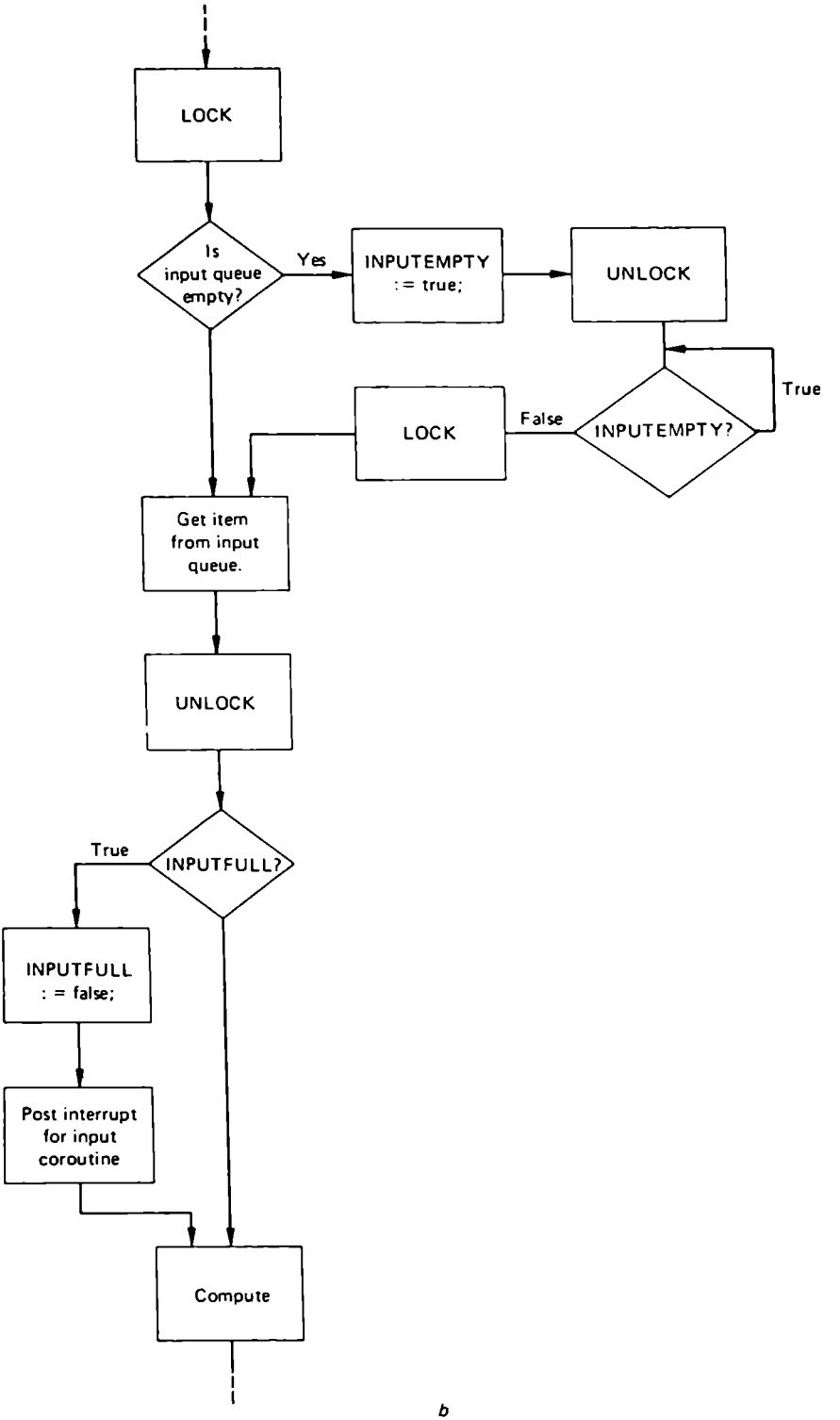
The LOCK and UNLOCK statements required for manipulating the output queue are essentially identical to the ones shown in Figure 8.15 for the input queue, so they are not shown here.

The reader should verify for himself that it is not necessary to turn off interrupts while performing the SIGNAL processes in Figure 8.15. In more general situations it may be necessary to turn off interrupts for the entire three-step sequence of WAIT, queue manipulation, and SIGNAL. A careful analysis is required to ascertain how much of a coroutine can be left UNLOCKed, but the conservative approach should be to LOCK statements where the analysis is difficult.



a

FIGURE 8.15 (*a*) An input coroutine with LOCKs and UNLOCKs



b

FIGURE 8.15 (*Continued*) (*b*) A main program with LOCKs and UNLOCKs surrounding its input processing

The LOCK and UNLOCK statements can be implemented on the PDP-11 computer as follows. We take advantage of the fact that the program status word of a program is treated as memory location 177776_8, so that we can modify priority by modifying the contents of this memory address as shown below.

```
LABEL     CODE   OPERAND     COMMENTS
PS        =      177776      ; Address of PS.
LOCK:     MOV    PS,TEMP     ; Save priority of current program.
          BIS    #340,PS     ; Raise priority of program to highest
                             ; priority (seven) to prevent interrupts.
UNLOCK:   MOV    TEMP,PS     ; Restore priority of initial program to
                             ; PS.
```

We now return to the situation, described in connection with Figure 8.13, in which one coroutine posts an interrupt for another. This is called a *software interrupt*. If a computer does not have instructions to force one, a software interrupt can be implemented simply by performing the same operations as for a hardware interrupt. Recall that, upon receiving an interrupt, the PDP-11 computer stores the P and PS registers on the machine stack and loads the new P and PS from the interrupt vector of the device. Example 8.9 simulates an interrupt to the input coroutine from the main program output section. DEVICE is the address of the input device's interrupt vector.

EXAMPLE 8.9

This sequence of instructions simulates an interrupt to reawaken the coroutine associated with DEVICE.

```
    LABEL     CODE   OPERAND          COMMENTS
1.  PS        =      177776
2.            MOV    PS,-(SP)         ; Push the current PS onto
                                      ; the stack.
3.            MOV    RETNAME,-(SP)    ; Push the address of RE-
                                      ; TURN onto the stack.
4.            MOV    DEVICE+2,-(SP)   ; Push the PS of the DE-
                                      ; VICE interrupt vector on-
                                      ; to the stack.
```

```
    LABEL       CODE    OPERAND          COMMENTS
5.              MOV     DEVICE,—(SP)     ; Push the program coun-
                                         ; ter onto the stack.
6.              RTI                      ; A return from simulated
                                         ; interrupt.
7.  RETNAME:            RETURN           ; Address of RETURN.
8.  RETURN:                              ; The main program will
                                         ; continue from here.
```

To demonstrate that the above segment of code does simulate an interrupt, consider the following. Assume that the input/output coroutines follow the conventions of Example 8.7. The three program stacks are configured as shown in Figure 8.16*a* with the main program executing and the input/output coroutines awaiting an interrupt. SP represents the contents of REG[6]. MSP, INSP, and OUTSP are memory cells containing the top of stack pointers for the main program, input program, and output program, respectively. Assume the input device causes an interrupt. At an intermediate point in the input coroutine the stacks appear as shown in Figure 8.16*b*.

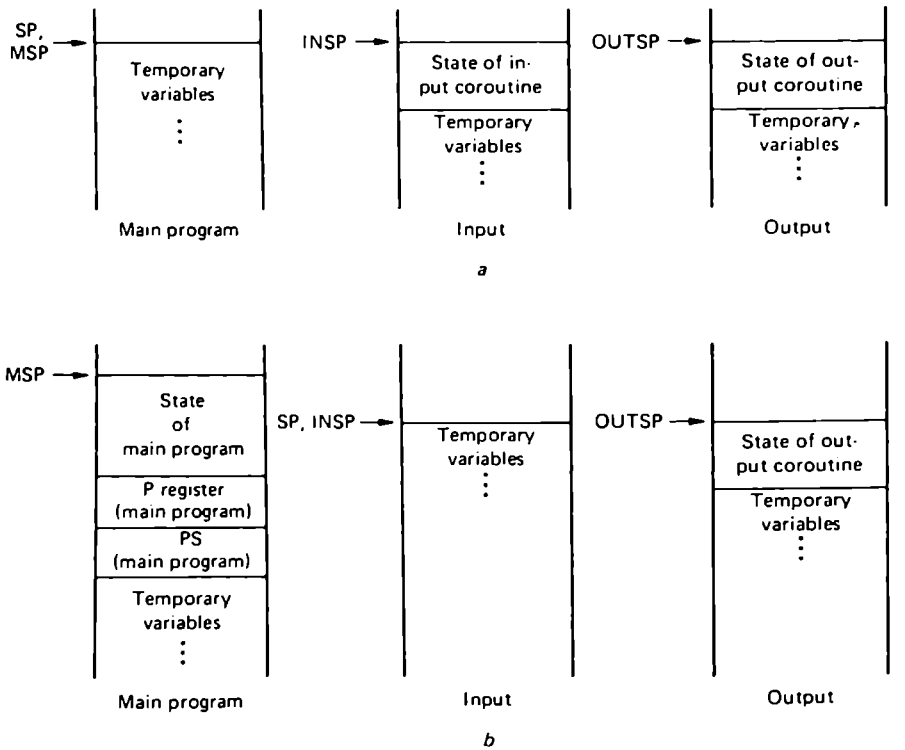


FIGURE 8.16 The coroutine stacks with (*a*) main program executing and (*b*) input coroutine interrupting

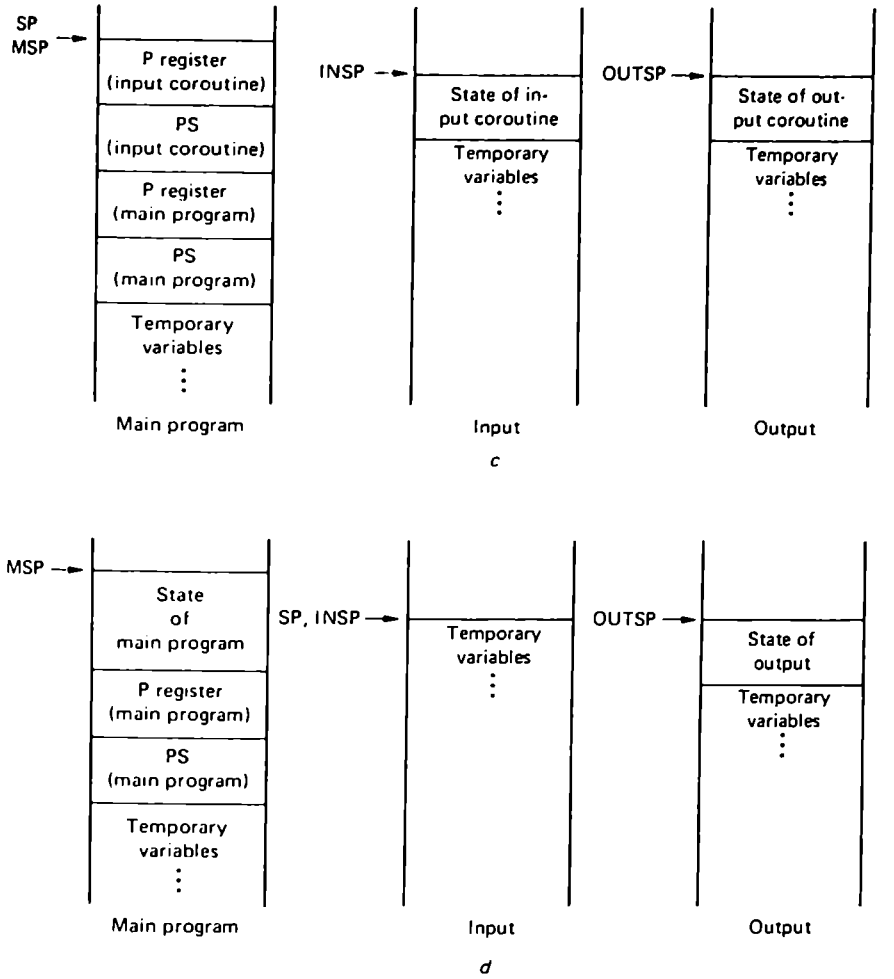


FIGURE 8.16 (*Continued*) The coroutine stacks when simulating a software interrupt to the input coroutine (c) just before the RTI and (*d*) during execution of the input coroutine after entry with RTI

Now consider the case in which the input coroutine is suspended. The three stacks, just before execution of the RTI in the software interrupt, are shown in Figure 8.16c. After executing the RTI, and while executing the input coroutine, the stacks are as shown in Figure 8.16*d*. This is identical to the situation that occurs when the input device actually causes an interrupt. Hence the software interrupt actually does simulate a hardware interrupt.

The techniques discussed in this chapter will work correctly for a large class of problems on a large number of different computers. Nevertheless, careful and thorough analysis is essential for constructing input/output programs, because a situation may present special problems that do not fit the framework of our discussion.

REFERENCES

Nearly all the earliest computers had some capacity for performing input/output concurrently with computation. Leiner (1952), for example, wrote an early article describing the hardware in an interface in great depth, but he was by no means the first designer to conceive of concurrent input/output. Burks, Goldstine, and Von Neumann (1946), in their proposal to build a computer, mentioned the value of concurrent input/output and the fact that it could be achieved with buffer storage and synchronization logic. They elected to leave it out of their initial designs because of the relatively high cost of hardware at that time. However, four years after their proposal, computers with concurrent input/output ability were under construction.

Interrupts and interrupt-driven input/output programs must also be credited jointly to many of the pioneers in computing. Much of the development of sophisticated interrupt-driven input/output programming techniques was spurred by the wide acceptance of the IBM 704 (later the IBM 709) in the middle and late 1950s. An outgrowth of a project initiated by the SHARE users' group for this computer was an input/output subroutine library that used interrupts and queues similar to those described here. A description appears in Mock and Swift (1959). In Europe, Dijkstra (1958) developed comparable programs for which he deserves equal credit. Recent studies of the control of concurrent input-output operations in the context of multiprocessing systems appear in Lampson (1968), Dijkstra (1968), and Wirth (1968).

EXERCISES

8.1 Write a program in assembly language for the four processes below. Use four different entry points to the program and employ a common collection of input/output subroutines. The interrupt system need not be used in this program.

a. Simple teletype echo. Characters keyed on the teletype are echoed by the computer and printed on the keyboard.

b. Queued teletype. Characters keyed on the teletype are queued and not echoed. Each time the character Z (332_8 or 132_8) is received, it is queued and the five characters at the head of the queue are printed. If fewer than five characters are in the queue, then just these characters are printed.

c. Push-down stack echo. Characters are keyed on the teletype and entered into a push-down stack. When the character *Z* is struck, the stack is emptied in last-in-first-out order.

d. Coroutine echo program. Use coroutine to expand number-letter pairs to *n* occurrences of the letter, where *n* is the value of the number. That is, "4F" becomes "FFFF." Use the coroutine in a simple teletype echo program, so that the echoed characters are expanded forms of the keyed-in characters. Only the numbers 2 through 5 should cause expansion. The least significant seven bits of the code for these numbers are 062_8 to 065_8.

8.2 Write an assembly language program for the SAVE and UNSAVE operations. Assume that REG[0] and REG[1] of the main program must be saved and that REG[0] of the input/output program must be saved.

8.3 Write a program that uses the interrupt system on the PDP-11 computer and does the following:

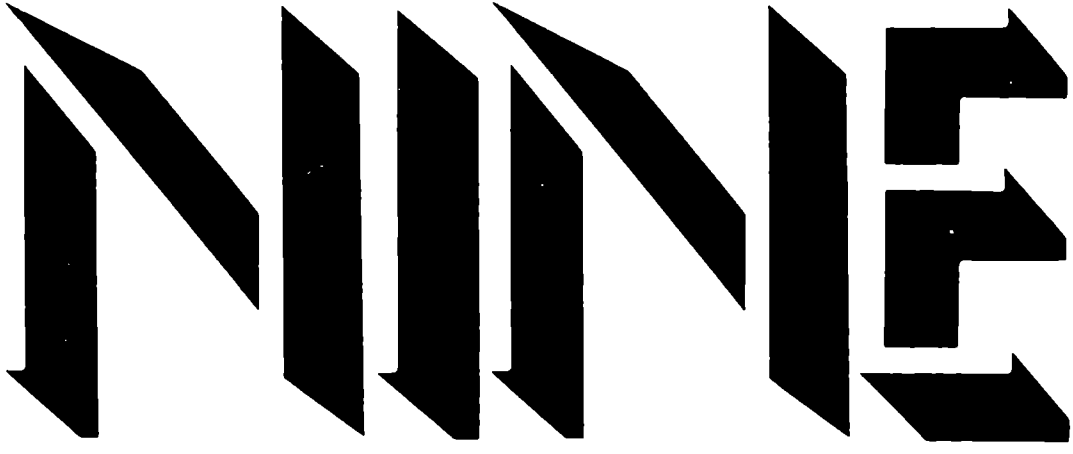
a. Characters are read from the paper-tape reader and queued.

b. Characters are removed from the queue and output to the teletype.

c. The main program (after initialization) is a single instruction, SELF: BR SELF.

8.4 Show that the coroutines of Figure 8.15 can never be suspended simultaneously.

8.5 Show that if the JSR RET,UNSAVE and the MOVB OUTDATA,PPB instructions (lines 17 and 18) of Example 8.7 are interchanged, then the program could fail. (Hint: What would happen if the device interrupted the UNSAVE routine at line 8?)


NINE

LINKED DATA STRUCTURES

Possibly the most important factor in the construction of efficient algorithms is the representation of data. Computer instruction repertoires are biased to perform some processes very efficiently and some inefficiently. For example, compare the difficulty of the following two computations:

1. Find the value of A[I], given the value of I.
2. Given the value of X, find a value of I for which A[I] = X, if there is such an I. (If there is more than one such I, pick any.)

Clearly the first computation requires only a simple index operation, but the second might require that the entire array A be searched. Consequently we should expect that the second computation will be considerably longer than the first.

The conclusion is not necessarily correct, however, because it presupposes that the A array is represented as a linear array as described in Chapter 6. Under certain conditions we can use a different representation for A[I] and perform the second computation with practically the same efficiency as the first. Suppose, for example, that X is an integer with a value constrained to lie in the interval $0 \leq X \leq N$, where N is not too large. Then we can use an array IVALUE, indexed from 0 to N, with the property that

If IVALUE[X] = I, then A[I] = X

and

If IVALUE[X] = $\varnothing$, then there is no I such that A[I] = X

Here the symbol ∅ is a special sentinel. We can choose ∅ to be an integer that is out of bounds for A. Note that IVALUE is another representation of the array A, but that some information about A is lost in this representation. If, for example, A[I1] = A[I2], then the representation we have permits us to deduce the value of at most one of the pair A[I1] and A[I2].

We need not keep A in its usual array representation if the only processing to be performed on it is the search described in the second computation above. Thus we see that it is possible to obtain rather large increases in processing speed by adapting the representation of a data structure to the processing to be performed on it. In this chapter and the next we shall explore some of the alternatives to the representations discussed earlier. Chapter 11 discusses several uses of these representations in algorithms for searching and sorting.

9.1 SIMPLE LISTS

The programmer frequently finds that he must organize his data into lists that are linearly ordered. If new items are added to one end of a list and no items are ever deleted, then the list can be implemented as a one-dimensional array, where the first element is A[1], the second is A[2], etc. When items are added to or removed from the list, and it is known that all operations must be performed at either end of the list, the appropriate implementation of the list is a push-down stack, a queue, or a double-ended queue, depending on the case at hand. If items are to be inserted or deleted somewhere in the interior of the list, then it is often best to abandon the notion of contiguous storage of items in the list. The difficulties in inserting and deleting items in the interior of lists are illustrated by the following example. Suppose that a list with MAX items is to be stored in a one-dimensional array named A. Then to delete the tenth item in the list we might execute the program

```
for i := 10 step 1 until MAX−1 do
    A[i] := A[i+1];
```

Similarly, to insert an item as the new tenth element of the list we might execute the program

```
for i := MAX step −1 until 10 do
    A[i+1] := A[i];
A[10] := newitem;
```

The time penalty for executing the insert and delete programs becomes excessive when MAX is large. Moreover, a great deal of memory can be wasted if the list that is actually stored in A contains relatively few ele-

ments in comparison to the size of the array. The difficulty in inserting and deleting is a result of contiguous storage of elements in the list. It is possible to improve matters substantially by using noncontiguous storage.

To see how list items can be held in noncontiguous storage we shall make use of two arrays: one to hold the items in the list and the other to point to the next items in the list. Suppose the item array is called V for value and the pointer array is called NEXT. We let NEXT[I] be the index in V of the item that follows V[I]. The pointer value 0 can be used to denote the end of a list, provided that V has lower bound 1—that is, that V[0] is not an element of V. The following program can be used to scan a list from V[start] to the end of the list, where the procedure LOOKAT is assumed to process an item in a list:

```
I := start;
loop: LOOKAT(V[I]);
    I := NEXT[I];
    if I ≠ 0 then go to loop;
```

We call this representation a *linked list*, and we sometimes use *link* interchangeably with *pointer*. The program demonstrates that a linked list can be scanned from one end to another as easily as a simple array can be scanned. We now show that items can be inserted and deleted in linked lists rather easily. The two procedures in Example 9.1 illustrate the processes; Figure 9.1 shows the processes graphically.

EXAMPLE 9.1

```
procedure insert(item,V,NEXT,place);
array V,NEXT; integer item, place;
comment V[item] to be inserted into the list as the successor of V[place];
    begin
        NEXT[item] := NEXT[place];
        NEXT[place] := item;
    end procedure insert;
procedure delete(place,V,NEXT);
array V,NEXT; integer place;
comment the successor of V[place] is deleted from the list;
    begin
        NEXT[place] := NEXT[NEXT[place]];
    end procedure delete;
```

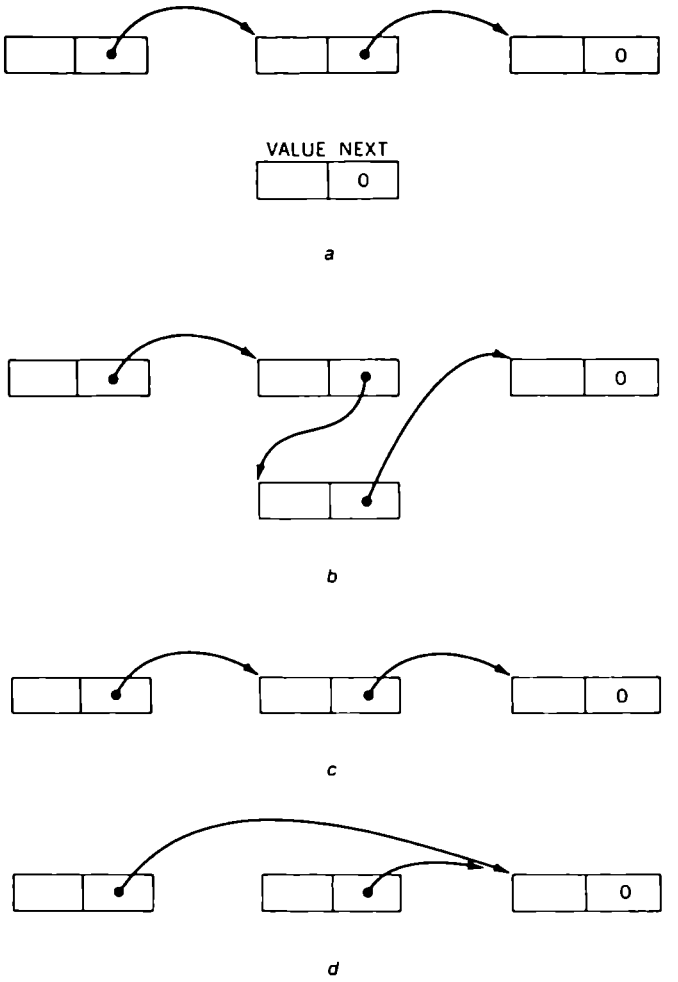


FIGURE 9.1 List insertion and deletion: (*a*) and (*b*) insertion; (*c*) and (*d*) deletion

It is somewhat redundant to use the array V as a parameter for the procedures in Example 9.1, because it is never referenced. The reason for placing it in a prominent position is to emphasize its relation to the array NEXT. In actual practice the arrays V and NEXT are often one and the same. Data packing is used to place a pointer to a successor element in the same word that contains the value of an element of the list.

To illustrate applications of linked lists let us consider the use of the queue for input/output. For computers such as System/360-370, which use data channels for input/output, the input/output requests for particular channels have to be queued, since the data channels can be busy when a request is made. If a first-in-first-out policy for honoring requests

is in effect, then a queue of the type described in Chapter 6 is adequate. In many cases performance is significantly better if the queue is ordered on the basis of priority or some other attribute instead of on a first-in-first-out basis. The performance improvement is usually related to the ability to schedule requests to minimize the rotational delay of moving magnetic memories such as drums and disks.

For this problem let us compare the efficiency of the linked-list representation of a queue with that of the array representation in Chapter 6. Assume that a request is received which belongs in the queue between items I and I+1 and that the queue contains N items. The REQUEST processing algorithm breaks naturally into two parts: search for the place to enter the item, then item entry. If we use the array representation of the queue and search the queue sequentially, beginning at the tail, then the search time will be approximately the same as the time required to search a linked-list representation of the queue, beginning at the tail. How do we make an insertion in the middle of the array representation of a queue? We do this by moving the items in the queue to make space for the new request. We may have to move as much as half the queue to do this. However, with a linked-list representation the insertion can be made by moving a few pointers. The time required for this is independent of the length of the list. Thus the linked-list representation of the queue is more efficient for inserting requests than the array representation. We shall have more to say about this later.

Another important application of the linked list is text editing. Processing of ordinary text by computers has become rather common as computers have become larger, faster, and more economical to operate. Consider the representation of a moderate-sized document in a computer memory, and assume that we wish to change all occurrences of beta in the text to alpha. If we store the text in an array representation, with each array element packed with characters, then we cannot replace the four characters in beta by the five characters in alpha without moving some text to create space. Note that we have to scan the array from beginning to end to find all occurrences of beta, and so we cannot take advantage of the efficient indexing that is characteristic of array representations.

Consider the same problem when the representation is a linked list, with each item in the list a character of text. The search for each occurrence of beta takes about the same amount of time in each representation, but when a text replacement is made, the linked list leads to considerably more efficient processing. Of course, the linked list requires extra memory for the pointers, but we do not have to use one pointer per character. For efficient storage the value field of a typical item in our linked-list representation might contain, say, up to 256 characters, and it would also

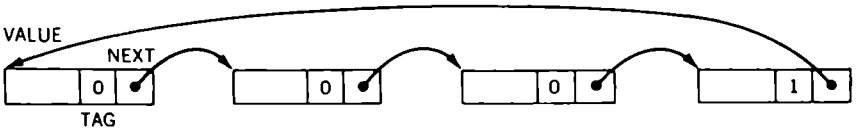


FIGURE 9.2 A circular list

contain an integer that gives the number of characters in the value field. In this representation we can replace beta by alpha everywhere and never have to move more than 255 characters for any replacement. As we increase the number of characters that we permit in a single value field, the memory used for pointers decreases and the processing time for a replacement increases. We clearly have a trade-off here in which we can exchange memory space for processing time.

The simple lists described here are frequently called *one-way linked lists* because they can be scanned only from beginning to end. Given a pointer to an element in the list, it is not possible, in general, to find the predecessor of the element. A simple generalization of the one-way linked list is the circular list shown in Figure 9.2. The last link in the list points back to the beginning of the list, instead of indicating no link as the 0 link does. It is usual practice in the implementation of circular lists to use a special tag bit in each link to identify the end of a list. If this bit is 0, for example, then the link is a true link to the next element. A tag bit of 1 indicates there are no more elements on the list, and the link points to the beginning of the list.

9.2 TWO-WAY LINKED LISTS

When lists are manipulated in such a manner that they must be scanned in both directions, the one-way linked-list structure is unsuitable, and the circular-list structure requires excessive processing to find the predecessor of an element in a list. The concept of linked lists is easily generalized to data structures that have bidirectional links. An example of such a structure is shown in Figure 9.3. Elements in such a data structure can be inserted or deleted by generalizing the basic procedures for scanning linked lists. The insertion process, for example, generalizes to the procedure of Example 9.2.

For application of the two-way linked list let us look again at the queue. We have already noted that one-way linked lists are useful when

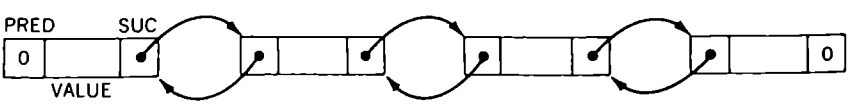


FIGURE 9.3 A two-way linked list

EXAMPLE 9.2

The following program shows how to insert into a two-way linked list:

```
procedure insert(item,V,SUC,PRED,place);
array V,SUC,PRED;
Integer item, place;
comment V[item] is to be inserted into the list as the successor of V[place],
SUC[place] is the link to the successor of V[place] in the list and PRED[place]
is the link to the predecessor of V[place] in the list;
begin SUC[item] := SUC[place];
    PRED[SUC[place]] := item;
    comment the two preceding statements changed the links that establish
    the correct successor to V[item], the next two statements establish the
    links for the predecessor of V[item];
    SUC[place] := item;
    PRED[item] := place;
end procedure insert;
```

items are to be inserted into the interiors of queues. Suppose that the insertion process requires that the insertion point be found by a *tail-to-head search*, and that each deletion requires that an item be found by searching the queue from *head to tail*. That is, we have to be able to search a queue in both directions. Since a one-way linked list cannot be scanned efficiently from head to tail, the one-way linked-list representation of a queue is inherently inefficient for this problem. However, the two-way linked list can be scanned in both directions efficiently, and thus it is satisfactory for this case.

Although two-way linked lists afford certain advantages in processing, we cannot disregard their disadvantages. Insertion and deletion are twice as costly as with one-way lists, because twice as many pointers must be altered to perform each operation. Moreover, the extra set of linkages in two-way linked lists occupies a significant amount of storage. Therefore, the usual practice is to use two-way lists only if list manipulation requires two-way scanning. Otherwise, one-way pointers are used.

There is an interesting case in which two pointers can be stored in the same number of bits that one pointer would normally occupy. This case is characterized by the fact that whenever an item is accessed, it is accessed by scanning the doubly linked list in one direction or the other. Thus when we reach a cell in the list, we know the address either of its predecessor or of its successor. The link in the cell need only contain the

value SUC+PRED, where SUC and PRED are the addresses of the successor and the predecessor, respectively. When we scan from the front of the list to the end, the value of PRED is known at each step, and the value of SUC can be determined by subtracting PRED from the link contained in the cell. Similarly, PRED can be computed if SUC is known. Thus we have the essentials of a doubly linked list.

Note that the link stored in each cell in this case must be one bit longer than the link for either PRED or SUC, because the maximum value for this link is twice the maximum value of PRED or SUC. We can even eliminate that bit if we choose to store the EXCLUSIVE OR of SUC and PRED instead of this arithmetic sum. If we let $\oplus$ denote the EXCLUSIVE OR operation, then we have the identities

$$\text{SUC} = \text{PRED} \oplus (\text{PRED} \oplus \text{SUC})$$
$$\text{PRED} = \text{SUC} \oplus (\text{PRED} \oplus \text{SUC})$$

Obviously we can recover the value of SUC if we know the value of PRED and of PRED⊕SUC.

9.3 THREADED LISTS

The various types of linked-list structures discussed thus far are primitive examples of linked data structures. Data structures of arbitrary complexity can be created merely by adding additional pointers to items on a list. The two-way list uses two pointers, one for successor and one for predecessor. This can be generalized, for example, to the representation of items in a two-dimensional matrix by using pointers to each of the four neighbors for each element in the matrix. In this case each item has four links associated with it, each of which "threads" the list elements in a particular direction—whence the name *threaded list*.

To illustrate this concept, suppose that we want to construct a list of people in several generations of a family so that we can deduce a number of family relationships from the data structure. To do this we can use the data structure shown in Figure 9.4. The name of each person has five pointers associated with it: father, mother, next oldest brother (or sister), oldest son, and oldest daughter. Figure 9.5 shows a family tree and its representation in this data structure. The sons of any person can be found by tracing the link from that person to the oldest son to find the first son; all other sons are then found by continuing to trace along the next oldest brother link until an empty link is found, How might we find the nephew, the grandchild, or the cousin of a person in the list? Note that we cannot find a person's spouse when there are no children.

Compare this example of a threaded-list representation to the other

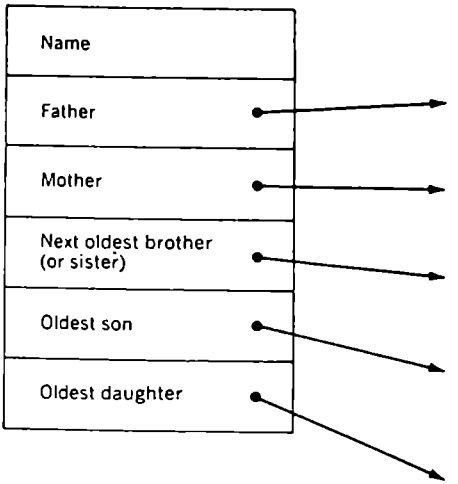
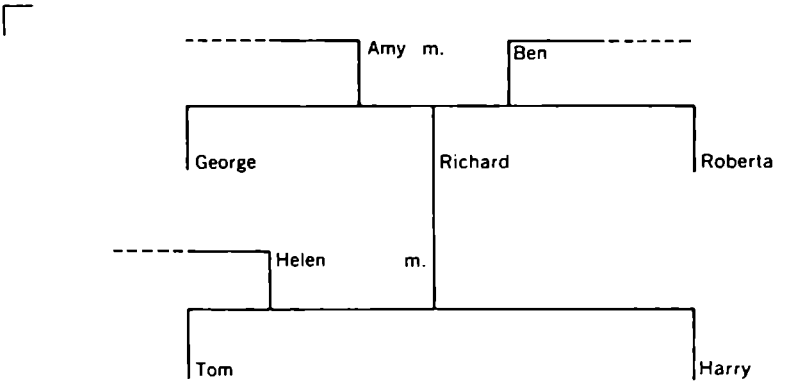


FIGURE 9.4 A threaded-list representation of a person and his family relationship

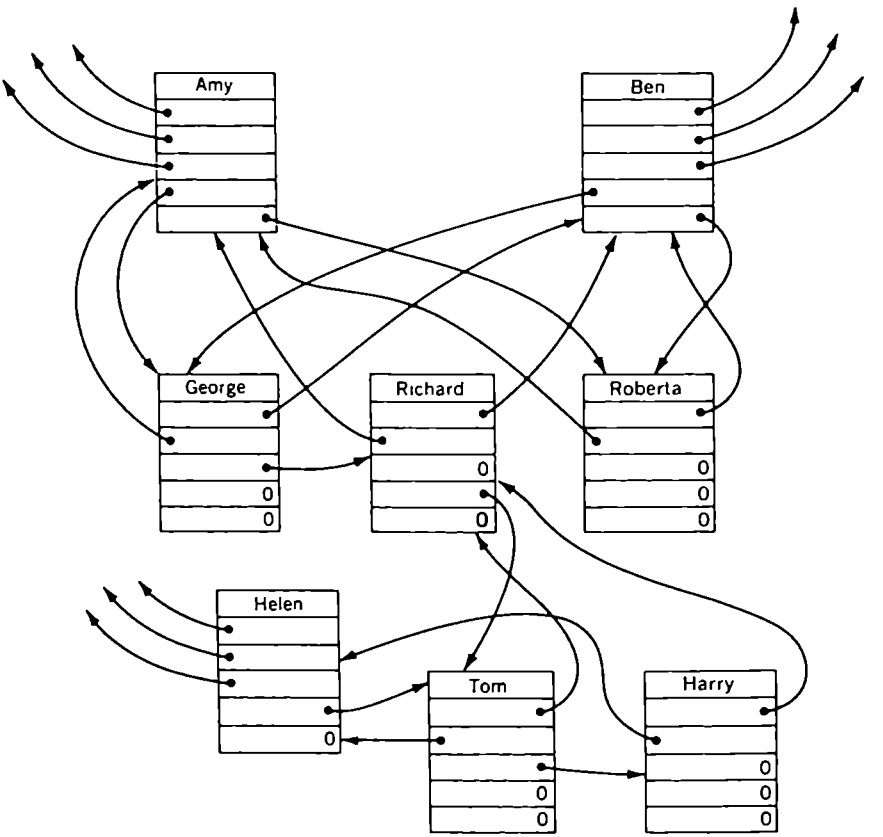
representations we have discussed. Can we represent a family tree in an array without pointers and still process the data efficiently? Suppose, for example, that we use a representation in which the data for each person are stored in a group of memory cells. In the first cell we put the person's name—say, JOHN SMITH. In the next cell we put his father's name, HENRY SMITH. The cells that follow then contain the names of his mother, his next oldest brother, etc. To identify his paternal grandfather, we look for his father's name and find HENRY SMITH. Now we have to look for the father of HENRY SMITH. However, this necessitates searching the entire array, since, by assumption, we have no pointers in the data structure. Obviously we want to keep the index to the data for HENRY SMITH, but this index is a pointer, and the use of such pointers creates the data structure shown in Figure 9.5.

There are many practical applications of threaded lists. One that arises frequently has to do with data that are ordered in two different ways. For example, organization membership lists are ordered alphabetically by name of subscriber, so that membership directories can be constructed efficiently, and by ZIP code because of Post Office requirements on organization mailings. We can use a threaded-list representation that satisfies both requirements. We use two threads, one that links members in alphabetical order and another that links members in order by ZIP code. For our processing requirements each thread can be a one-way linked list. Other aspects of the problem could lead us to consider circular and two-way linked lists instead of or in combination with one-way linked lists.

The many variations possible for threaded-list data structures are



a



b

FIGURE 9.5 (*a*) A portion of a family tree and (*b*) its threaded-list representation

limited only by the imagination. Linkages can be placed in a data structure freely, but they should be included only if they increase the efficiency of computation. The gain in computation speed must always be balanced against the increased use of memory and the added cost of inserting and deleting items. In Chapter 11 we shall discuss some tools for analysis and design of linked data structures.

9.4 MEMORY ALLOCATION AND GARBAGE COLLECTION

In the discussion of linked data structures the assumption has been implicit that unused memory is always readily available for new items to be linked into lists. However, when we delete an item from a list, the memory in which the deleted item had been stored is lost unless we take measures to reuse it. In practice we can seldom afford to cast away memory for deleted items. Available memory is usually far too small to hold all the active data lists plus all the items that have ever been deleted from them.

Early list-processing programs utilized the list mechanism itself to control available memory and to recover memory made available by deletions. All available memory was linked to a list known as the *free-storage list*. When items were deleted from active lists, the unused memory was inserted at the beginning of the free-storage list. Recovery of memory in such programs was nearly always specified explicitly by the programmer.

In more recent programming systems, such as LISP, a method known as *garbage collection* is used. All active lists declared in these languages are linked together, so that at any point an executive program can trace links and determine all active memory locations. Then, when available space is exhausted, a garbage-collection program is called to do the link tracing. As this program scans the links, it tags each memory location with a special bit if it is an element of an active list. All untagged locations are then known to be garbage and are reclaimed into a free-storage list. The garbage-collection process is invisible to the programmer.

REFERENCES

Although many of the early computer scientists deserve to share credit for linked data structures, the most extensive work was done by Simon, Newell, and Shaw. They originated a novel programming language called IPL-V, in which the only data structures available were linked

lists and push-down stacks (cf. Newell and Shaw, 1957; Shaw et al., 1958; Newell, 1961). This language greatly influenced subsequent languages invented for manipulating nonnumerical information. Current trends in language development suggest that the notion of linked data structures is too fundamental to be compartmentalized in the field of nonnumeric processing. Recent languages, such as PL/I and the proposed successors to ALGOL, include capabilities for building linked lists and presume an underlying garbage-collection processor for storage reclamation (Wirth and Hoare, 1966; Van Wijngaarden et al., 1968). Ross's article on data structures, which he called *plexes*, marks one of the early references to threaded lists (Ross, 1961).

Techniques for storage management and garbage collection in programming systems that use linked data structures have been developed in parallel with the linked data structures. Simon, Newell, and Shaw are usually credited with the idea of placing all unused memory on a list from which cells can be allocated as needed. IPL-V did not have an automatic garbage-collection algorithm, but depended on the programmer to return each cell to the free-storage list as it became available. An excellent description of garbage-collection algorithms, the subtle constraints on them, and the history of their development appears in Knuth (1968).

EXERCISES

9.1 Let X be a large two-dimensional array—say, 1000 × 1000—and assume that it is sparse in the sense that very few of its elements are not 0. Construct a data structure for X that requires an amount of memory proportional to the number of nonzero elements of X. Write a subroutine in an ALGOL-like language for finding the value of X[I,J], given I and J.

9.2 Let X be a given number, and let L be a list of numbers one of which is X. If L is represented as a one-way linked list, and it is equally likely that X is any one of the elements of L, then what is the average number of elements that must be scanned to find which one is X? Find the average when X has probability p of being on the list and probability $1 - p$ of not being on the list. Assume that if X is on the list, it has equal probability of being anywhere on the list.

9.3 Let L be a one-way linked list. Write subroutines in an ALGOL-like language for appending a new element to the end of the list and for creating a new one-way linked list that contains the elements of L in reverse order.

9.4 Construct a linked-list representation of a queue. Write subroutines in an ALGOL-like language for PUT and GET processes for the queue.

9.5 Let L be a list such that each element of L is itself a list. We call the second-level lists *sublists* of L. Write a subroutine in an ALGOL-like language that constructs a new list which contains a copy of every item that is on two or more sublists of L.

9.6 Repeat Exercise 9.5, but instead of creating a new list, use threaded links to link together every reference to the same item.

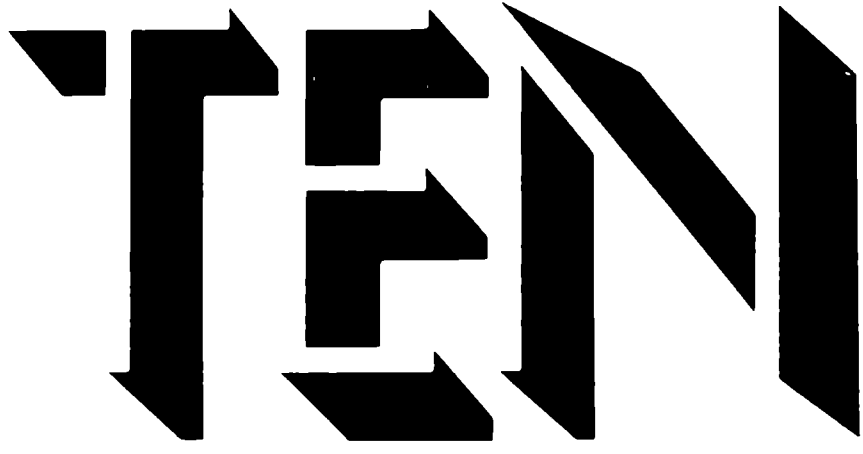
9.7 Suppose it is necessary to be able to add items efficiently to the head of a list, as well as to the end. Show that this can be done efficiently with one-way linked lists by maintaining two pointers to the list, one to the head and one to the tail.

9.8 Let L be a list, and let each of its elements be a list that contains numbers from the set $\{1, 2, \ldots, N\}$. Assume that no number appears on more than one sublist of L.

a. Write a subroutine in an ALGOL-like program that merges two specified sublists of L into a single sublist of L (L should have one less sublist after the merge operation).

b. Write a subroutine that can determine, for a specified pair of sublists, if there exists an integer I on one sublist and an integer I+1 on the other.

c. Write a subroutine that adds a threaded link to L, beginning with the number 1 and tracing through the integers in numerical order.


TEN

ROOTED TREES, RECURSIVE PROGRAMS, AND PUSH-DOWN STACK MACHINES

The title of this chapter suggests a composite discussion of three widely separated topics. Actually these three topics are quite closely related. The rooted tree is a graphic flow diagram of a recursive program, and the stack machine is the natural vehicle for executing the recursive program.

10.1 ROOTED TREES

Rooted-tree data structures are generalizations of linear arrays and linear linked lists. Rooted trees are useful in many algorithms for which the linear data structures do not support efficient processing. In this section we shall examine some of the properties of such trees and discuss methods for using them in data structures.

Figure 10.1 exemplifies the class of graphs called *rooted trees*, informally defined as follows:

> **Rooted Tree** A graph that consists of a collection of nodes and branches, with each branch connecting two nodes. The branch relates the two nodes at its ends such that the node at the upper end of the branch is identified as a *predecessor* of the node at the lower end of the branch. If node x is a predecessor of node y, then node y is said to be a *successor* of node x.

A rooted tree also satisfies the following properties:

1. There is a unique node (the *root*) which has no predecessor.
2. Every node other than the root has precisely one predecessor.

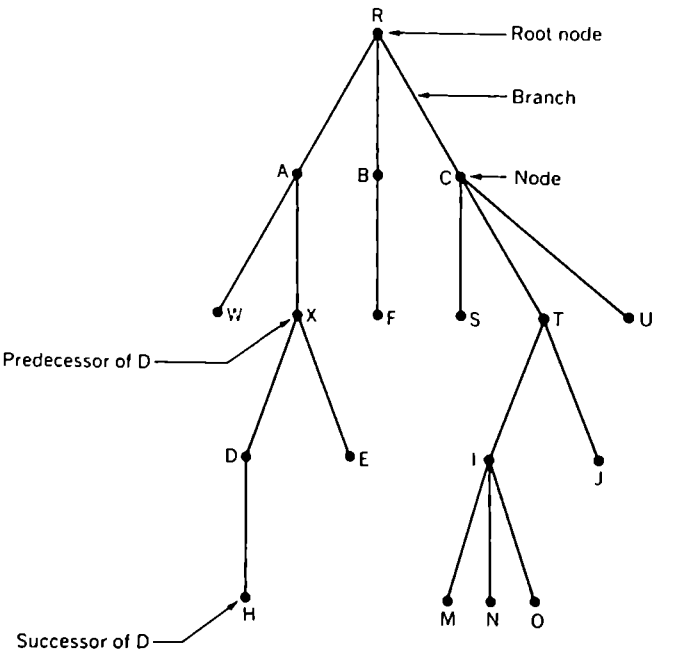


FIGURE 10.1 A typical rooted tree

3. Every node other than the root is connected to the root by a (unique) path such that the path begins at the root and ends at the node, and such that each node on the path other than the root is a successor of the previous node on the path.

There are no closed loops in rooted trees. For the remainder of this chapter we shall use *tree* to mean rooted tree.

In the tree shown in Figure 10.1 the successors of node T are nodes I and J, and the predecessor of node T is node C. The unique path from the root, node R, to node T is the path R C T.

We have had occasion to make use of trees in earlier chapters. Recall, for example, the data structure for multidimensional arrays using indirection. An example of interest to us is the representation of arithmetic expressions in tree form, as illustrated in Figure 10.2. In forming these graphs we use interior nodes (nodes with successors) to represent the operations $\{+, -, *, /\}$, where each of the successors of an interior node is an operand. Note that some operands are themselves expressions. The leaves of the trees (nodes without successors) are simple variables indicated by A, B, C, ...

Note that each interior node is the root of a tree that consists of the interior node, its successors, the successors of its successors, their successors, etc. We call such a tree a *subtree* of the original tree. There are also subtrees associated with each *leaf* in the tree, namely the subtree that consists of just a leaf by itself.

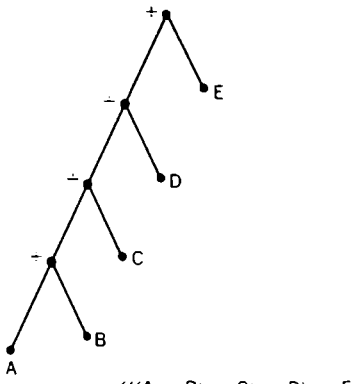
With each node is associated an integer called the *level* of the node, which is defined to be 1 greater than the length of the unique path in the tree that is directed from the root to the node. By definition, the level of the root is 1, the level of its successors is 2, the level of the successors of the successors is 3, etc. In Figure 10.1 node R is at level 1 and node X is at level 3.

The *degree* of a node is the number of successors of a node. Thus the leaves of a tree have degree 0. In Figure 10.1 node I has degree 3 and node T has degree 2.

An interior node of a tree is called a *branch node*. If a tree has n nodes, then it has $n - 1$ branches. This is because every node except the root has a predecessor, so that there are $n - 1$ nodes with predecessors in the tree. Each predecessor-successor pair is identified with a unique branch.

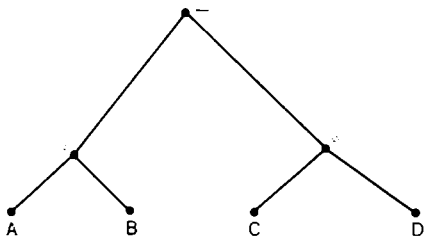
An *ordered tree* is a tree in which the successors of each node are ordered. For example, in Figure 10.1, if we order the successors of a node from left to right; then node S is the first successor of node C, node T is the second successor, and node U is the third successor.

A special class of trees that is very important for the representation



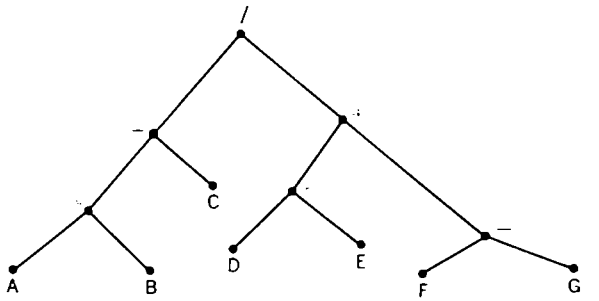
(((A × B) ÷ C) − D) × E

a



(A B) − (C D)

b



((A − B) × C)/((D E) ÷ (F − G))

c

FIGURE 10.2 Tree representations of arithmetic expressions

of data is the class known as *binary trees*. Specifically, a binary tree is a rooted tree in which every node has no more than two successors. Moreover, each successor of a node is identified as a *left successor* or a *right successor*, and no node may have two left successors or two right successors. We draw binary trees by slanting branches to the left for left suc-

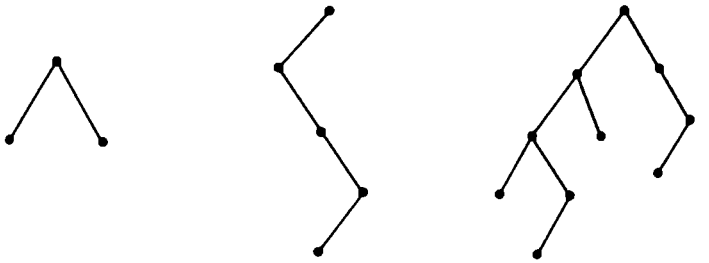


FIGURE 10.3 Examples of binary trees

cessors and to the right for right successors, so that the successor type can be determined by inspection of the drawing. Figure 10.3 shows examples of binary trees.

10.2 TRAVERSING BINARY TREES

A *traversal* of a data structure is a scan through the data structure such that each element is examined exactly once. If we consider trees to be a generalization of arrays, then we must find ways to traverse trees in much the same way that the following ALGOL statement allows us to examine each element of the array A:

```
for i := 1 step 1 until N do
    examine(A[i]);
```

An equally natural way to examine arrays is to start at N and index backward through the array.

The corresponding process for trees is best explained for binary trees. Later we shall see how any ordered rooted tree can be represented by a binary tree, so that the traversals presented here are completely general. This presentation follows that of Knuth (1968).

The natural methods for traversing a tree require that we break the tree into three disjoint components: the root, the left subtree of the root, and the right subtree of the root. Tree-traversal algorithms specify the order in which these components are traversed. Since each subtree of the root is again a tree, it too can be broken into a root and a left and right subtree, which are to be traversed in the same specified order. Thus subtrees are broken into smaller and smaller components, until every component has only one node. At this point we have determined the order in which the entire set of nodes is visited.

The first traversal algorithm, that for postorder traversal, can be informally stated as follows:

Postorder Traversal Visit the nodes in an order such that, for any node V in the tree, all the nodes in the left subtree of V are visited before V, and all the nodes in the right subtree of V are visited after V.

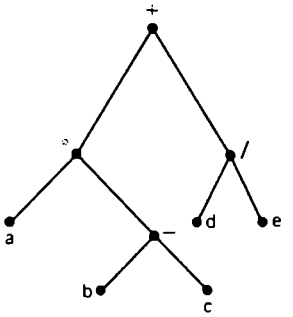


FIGURE 10.4 Tree representation of a*(b − c) + d/e

To traverse the tree in Figure 10.4 in postorder we visit all the nodes in the left subtree of +, then +, and then all the nodes in the right subtree of +. The nodes visited before + have to be visited in the order such that the nodes in the left subtree of * are visited, then *, and then the nodes in the right subtree of *. By following this line of reasoning, the postorder traversal of Figure 10.4 is found to be

$$a*b - c + d/e$$

To simplify the description of the postorder traversal algorithm, we write it as follows:

Postorder Traversal

1. Traverse the left subtree in postorder.
2. Visit the root.
3. Traverse the right subtree in postorder.

Two other traversal algorithms are of importance to us:

Preorder Traversal

1. Visit the root.
2. Traverse the left subtree in preorder.
3. Traverse the right subtree in preorder.

Endorder Traversal

1. Traverse the left subtree in endorder.
2. Traverse the right subtree in endorder.
3. Visit the root.

Traversal of Figure 10.4 in postorder, preorder, and endorder causes the nodes in the tree to be visited in sequence as follows:

Postorder	$a*b - c + d/e$
Preorder	$+*a - bc/de$
Endorder	$abc - *de/+$

In all three cases the relative order in which the leaves are visited is the same. The essential difference in the traversal methods has to do with the sequence in which the branch nodes are visited in relation to each other and in relation to the leaves. Preorder indicates that the root is visited before the left subtree, and postorder indicates that the root is visited after the left subtree. Of the three traversal schemes, the first two are the most important.

The algorithm in Example 10.1 traverses a tree in postorder. By assumption, the tree is represented in the array A, so that all the information about node i in the tree is stored in A[i] as packed data. The notation A[i].leftlink refers to the portion of the data in A[i] that is a link to the left successor of node i and is meant to suggest that the left successor link of node i will be found by unpacking the data in A[i] as described in Chapter 6. Similarly, A[i].rightlink is a link to the right successor of node i. A link in this example is the index of the successor in the array A. Thus if node 10 is a left successor of node 5, then A[5].leftlink has the value 10.

EXAMPLE 10.1

The procedure PUSHDOWN and POPUP are push-down stack procedures. POPUP has two parameters in this case, rather than one. The first parameter of POPUP receives a copy of the top of the stack before the stack is popped. The second parameter is a label to which POPUP returns if a stack underflow occurs when the stack is popped.

```
procedure postorder(A);
array A;
comment root of tree is in A[1], left subtree and right subtree are pointed to
by indices contained in A[1].leftlink and A[1].rightlink, respectively;
begin integer i;
    clearstack;
    comment the above procedure clears the push-down stack;
    i := 1;
```

```
checkrootempty:
    comment 0 denotes an empty link;
    if i = 0 then go to returntopredecessor;
    PUSHDOWN(i);
    i := A[i].leftlink;
    go to checkrootempty;
returntopredecessor:
    POPUP(i,exitifunderflow);
    examine(A[i]);
    i := A[i].rightlink;
    go to checkrootempty;
exitifunderflow:
comment when stack underflows algorithm terminates here;
end;
```

When the tree in Figure 10.4 is represented as shown in Table 10.1, the postorder algorithm produces the results shown in Table 10.2. Table 10.2 lists the state of the stack at the time each node is examined.

The reason Example 10.1 is a correct algorithm is that it reflects the recursive structure of the tree. The order in which nodes are processed is left subtree, root, right subtree. Since a subtree is itself a tree, the instructions for processing the subtrees are precisely the same as those for processing the tree that contains the two subtrees. Note how the PUSHDOWN and POPUP procedures are used to store and recover the pointer to the root. The pointers must be recovered in last-in-first-out order, so the stack is the appropriate data structure for storing the pointers.

The binary trees as we have represented them are a natural generali-

TABLE 10.1

Representation of the Tree in Figure 10.4

INDEX	NODE	LEFT LINK	RIGHT LINK
1	+	2	3
2	*	4	5
3	/	6	7
4	a	0	0
5	−	8	9
6	d	0	0
7	e	0	0
8	b	0	0
9	c	0	0

TABLE 10.2

State of the Push-down Stack During Execution of the Postorder Algorithm

i	EXAMINE(A[i])	STACK
4	a	2(*)
		1(+)
2	*	1(+)
8	b	5(−)
		1(+)
5	−	1(+)
9	c	1(+)
1	+	empty
6	d	3(/)
3	/	empty
7	e	empty

zation of the one-way linked list. In traversing trees it is the usual case to traverse them both from top to bottom and from bottom to top. Hence a more appropriate representation of a binary tree would be a generalization of the two-way linked list or of the circular list. In Example 10.1 we used the push-down stack to remember pointers that are needed when we traverse the tree in the opposite direction from the pointers. We can avoid the push-down stack by using special pointers, similar to those in circular lists, that point back into the tree. These special pointers allow us to traverse counter to the direction of the standard pointers. This pointer structure, credited to A. J. Perlis and C. Thornton (1960), is known as a *threaded tree*. Example 10.2 shows how to traverse a threaded tree.

EXAMPLE 10.2

```
procedure threadedpostorder(A);
array A;
comment A[1] contains the root. If A[i].lefttag is 1, then A[i].leftlink points to
the left subtree, otherwise the tag is 0 and the link points to the predecessor
of the node in postorder. Similarly, A[i].rightlink points to a right subtree or
the successor of A[i] in postorder;
```

```
begin integer i, savei;
    i := 1;
traceleft:
    if A[i].lefttag ≠ 0 then
        begin comment move down the chain of left successors;
            i := A[i].leftlink;
            go to traceleft;
        end;
check:
    if i = 0 then go to exit;
    comment 0 link indicates end of tree;
    examine(A[i]);
    savei := i;
    i := A[i].rightlink;
    if A[savei].righttag = 0 then go to check;
    comment 0 indicates a thread. In this case we have the correct successor;
    go to traceleft;
    comment 1 indicates normal link, follow chain of left successors;
exit:
end procedure threadedpostorder;
```

Figure 10.5 illustrates the data structure that is traversed. Links in the tree that would normally be empty are replaced by links into the tree, shown as dashed lines. Dashed lines pointing left are directed from a node n to the node that is the predecessor of n in postorder. Dashed lines pointing right point to the successor of n in postorder. For this procedure to recognize a halting condition, the rightmost dashed link in Figure 10.5

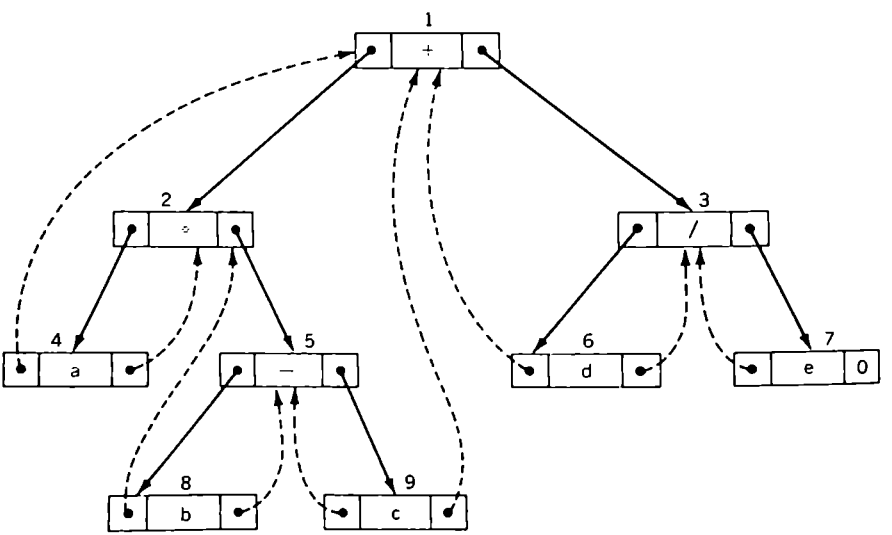


FIGURE 10.5 Threaded tree representation of Figure 10.4

is set to 0 and the leftmost dashed link points to the tree root. Dashed links are distinguished from ordinary links by examining A[i].lefttag and A[i].righttag. The 0 tag denotes a dashed link. If the tree is stored so that successors have higher indices than predecessors, then a magnitude comparison can be used instead.

Table 10.3 shows a data structure for the threaded tree in Figure 10.5, and Table 10.4 illustrates the values of intermediate variables in the threaded postorder procedure as it processes this tree. The values are shown at each point a tag is tested.

The importance of the schemes for traversing binary trees arises from the fact that binary trees can be used to represent arbitrary ordered trees. An example is shown in Figure 10.6. To be more precise, the representation is constructed to satisfy certain properties. If T is an ordered tree, then its representation as a binary tree T′ is as follows:

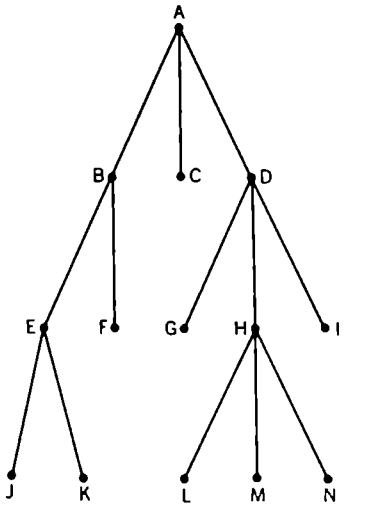
1. There is a one-to-one correspondence between the nodes in T and the nodes in T′.
2. The first successor of a node in T is the left successor of the corresponding node in T′.
3. The second to nth successors of a node N in T are linked in order in T′ as a chain of right successors. The chain begins at the node in T′ that corresponds to the first successor of N.

Thus to visit the successors of a node in the ordered tree when dealing with the binary representation of the ordered tree, we visit the left

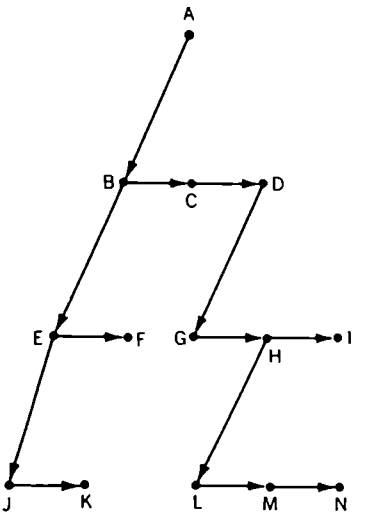
TABLE 10.3

Representation of the Threaded Tree in Figure 10.5

INDEX	NODE	LEFT TAG	LEFT LINK	RIGHT TAG	RIGHT LINK
1	+	1	2	1	3
2	*	1	4	1	5
3	/	1	6	1	7
4	a	0	1	0	2
5	−	1	8	1	9
6	d	0	1	0	3
7	e	0	3	0	0
8	b	0	2	0	5
9	c	0	5	0	1



a



b

FIGURE 10.6 Representation of an ordered rooted tree by a binary tree

successor of the node in the binary tree and then follow the chain of right successors. The binary-tree representation as described here is commonly used for list representations in the programming language LISP. We shall have more to say about binary representation of trees with respect to arithmetic expressions later in the chapter.

TABLE 10.4

Trace of the Execution of the Threaded Postorder Algorithm

i	LEFT TAG	RIGHT TAG	NEXT i	EXAMINE
1	1		2	
2	1		4	
4	0			a
4		0	2	
2		1	5	*
5	1		8	
8	0			b
8		0	5	
5		1	9	−
9	0			c
9		0	1	
1		1	3	+
3	1		6	
6	0			d
6		0	3	
3	1		7	/
7	0			e
7		0	0	

10.3 RECURSIVE PROGRAMS

Example 10.1 is an iterative program for an algorithm that is easily programmed recursively. In this section we shall explore the concept of recursion and then show methods for implementing recursion in computer programs. Finally, we shall relate the flow of control in recursive algorithms to tree structures. To clarify the concept of recursion let us first consider a recursive definition of a function, and then the recursive algorithm for computing this function (Example 10.3).

The term *recursion* indicates that the definition of the function f is defined to invoke properties of itself in an essential way. Recursive definition is valid if the definition includes a basis part which defines the function for some value of its arguments independent of the definition of the function for other arguments, and if the function for all other values of its arguments can be determined from the basis definition.

In the example above $f(0) = 1$ is the basis part of the definition. According to the recursive part, to find the value of the function for any

EXAMPLE 10.3

The recursive definition of $f(x)$ is

$$f(0) = 1$$

$$f(x) = \begin{cases} x \cdot f(x-1) & \text{for integral } x \geq 1 \\ \text{undefined} & \text{otherwise} \end{cases}$$

The ALGOL program to compute $f(x)$ is as follows:

```
integer procedure f(x);
integer x;
begin
    if x < 0 then f := ∞
    else
    if x = 0 then f := 1
    else
        f := x*f(x − 1);
end procedure f;
```

positive-integer argument we must know the value of the function for the next lowest integer. Thus to evaluate $f(x)$ for $x = 10$ we must know $f(9)$; this implies that we must know $f(8)$, etc., until we finally find that $f(1)$ is defined in terms of $f(0)$, which is defined by the basis part of the definition. In spite of the apparent circularity of the definition, we see that the function is truly well defined. What function have we defined?

For every recursive subroutine there must be some set of parameter values for which the subroutine does not call itself recursively. This corresponds to the basis part of a recursive definition. In Example 10.3 no recursive call occurs for $x = 0$. If x is greater than 0, the sequence of recursive calls decreases x until we find the value of $f(0)$, and then all the other values are defined in terms of $f(0)$. If a recursive subroutine calls itself regardless of the values of the parameters, then the subroutine can never reach its exit.

With respect to Example 10.1, we can trivially derive a recursive formulation for algorithms for traversing a binary tree in preorder, postorder, or endorder.

The procedure in Example 10.4 can be modified trivially for preorder traversal by exchanging lines 1 and 2 of the procedure body. An exchange of lines 2 and 3 yields a procedure for endorder traversal.

EXAMPLE 10.4

The following example illustrates tree traversal in postorder:

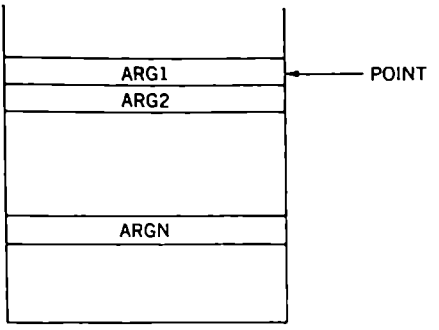
```
procedure postorder(i);
integer i;
comment A[i] is the root of a tree. Its subtrees are identified by A[i].leftlink
and A[i].rightlink;
begin
    if A[i].leftlink ≠ 0 then postorder(A[i].leftlink);
    examine(A[i]);
    if A[i].rightlink ≠ 0 then postorder(A[i].rightlink);
end procedure postorder;
comment to traverse a tree with root at A[1] use the call "postorder(1)" in
the calling program;
```

Now we come to the problem of the implementation of recursive procedures in assembly language. If we examine Examples 10.3 and 10.4 carefully, we note two potential problems that might arise in a direct translation to assembly language. One is that of saving the return address, and the other is that of saving parameters passed to procedures. Let us consider these problems separately.

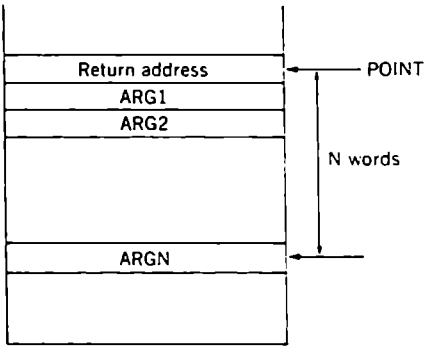
Since a recursive procedure can be entered two or more times without an intervening exit, and since the return addresses associated with the entries need not be the same, in general, we cannot use a single memory-storage location for holding return addresses. The natural storage mechanism for saving return addresses is the push-down stack, because the return addresses must be recovered in a last-in-first-out order.

Similarly, sets of parameters cannot be stored in an area that is just large enough to hold one parameter set, where by *parameter set* we mean a set of N values for a procedure that has N arguments. When a parameter set is passed to a recursive procedure it must be stored in a private area, because a recursive call will cause an entirely new set of parameters to be passed, and the new set cannot be allowed to destroy the old set. When a procedure that has been entered recursively reaches an exit, the newest set of parameters is thrown away, and the previous set of parameters becomes current. In fact the sets of parameters must be recovered on a last-in-first-out basis. Consequently, the push-down stack is the appropriate storage medium for passing parameters. The methods for using a push-down stack for parameter passing and storing return addresses were discussed in Chapter 7.

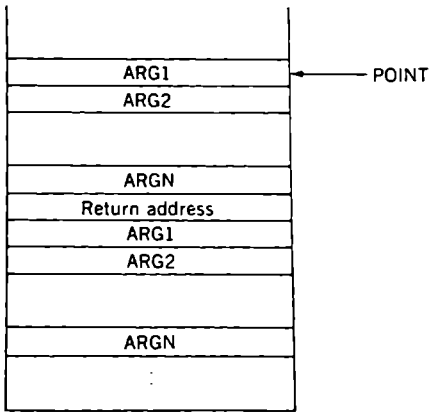
One last consideration is important for program implementation.



a



b



c

FIGURE 10.7 The state of the push-down stack for recursive subroutine RECUR: (*a*) just prior to entry point and (*b*) after saving return address. To access the *N*th parameter in this state decrement the stack pointer by *N*. (c) Parameters in stack for a recursive call

Recursive programs must not be self-modifying in such a way that program self-modification from one entry of the procedure causes the program to misbehave if a recursive entry occurs. This is generally taken to mean that recursive programs must never modify themselves—that is, recursive programs must be reentrant.

The following example shows canonical methods for implementing recursive procedures in assembly language for the PDP-11 and System/360-370 computers.

Figure 10.7 shows the state of the push-down stack during the execution of the instructions in Example 10.5. Just prior to subroutine entry, the top N items in the stack are the N parameters of the subroutine, with

EXAMPLE 10.5

The following instructions are in PDP-11 assembly language. POINT is the current pointer to the top of the stack.

```
LABEL    CODE   OPERAND              COMMENTS
POINT    =      %6                   ; The stack pointer.
P        =      %7                   ; The subroutine linkage reg-
                                     ; ister is specified as REG[7].
TEMP     =      %4                   ; A temporary register.
         . . .  . . .                . . .
         JSR    P,RECUR              ; Initial call on RECUR.
. . .                                . . .
RECUR:                               ; Entry to recursive proce-
                                     ; dure. The return address is
                                     ; stored on the stack.
         MOV    TWOI(POINT),TEMP     ; Load the Ith parameter.
                                     ; TWOI has the value 2*I.
         . . .  . . .                . . .
         MOV    ARGN,-(POINT)        ; Sequence of instructions for
                                     ; passing arguments in stack
                                     ; for a recursive call. Last
                                     ; argument stacked.
         . . .  . . .                . . .
         MOV    ARG1,-(POINT)        ; First argument stacked.
         JSR    P,RECUR              ; Recursive call.
```

```
LABEL    CODE    OPERAND          COMMENTS
         MOV     (POINT)+,%0      ; Preparation for exit. Pop up
                                  ; subrcutine linkage register.
         ADD     TWON,POINT       ; Pop stack N times. TWON
                                  ; contains 2·N.
         MOV     %0,—(POINT)      ; Restore subroutine linkage
                                  ; register to top of stack.
         RTS     P                ; Exit to address on top of
                                  ; stack.
```

The program below is the System/360-370 version of a recursive program. PUSH and POP are the stack subroutines, and both use REG[PARAM] for their parameter.

```
LABEL    CODE    OPERAND          COMMENTS
         ...
PARAM    EQU     1                The parameter register for PUSH
                                  and POP subroutines.
POINT    EQU     10               The stack pointer.
RET      EQU     14               The return-address register.
TEMP     EQU     6                A temporary register.
RECUR    LR      PARAM,RET        Entry point. Return address is in
                                  REG[RET], prepare to stack it.
         BAL     RET,PUSH         Push down contents of
                                  REG[PARAM]—that is, stack re-
                                  turn address.
...      ...     ...              ...
         L       TEMP,I           Beginning of a sequence of in-
                                  structions to access Ith param-
                                  eter.
         SLA     TEMP,2           Multiply I by 4, the number of
                                  bytes per full word.
         AR      TEMP,POINT       Add stack pointer; we now point
                                  to the Ith parameter.
         L       TEMP,0(TEMP)     Load the parameter into
                                  REG[TEMP].
...      ...     ...              ...
         L       PARAM,ARGN       Beginning of sequence of instruc-
                                  tions for a recursive call on RE-
                                  CUR, loads last parameter for the
                                  call.
```

LABEL	CODE	OPERAND	COMMENTS
	BAL	RET,PUSH	Push down last parameter.
...	...	...	...
	L	PARAM,ARG1	Load first parameter.
	BAL	RET,PUSH	Push down first parameter.
	BAL	RET,RECUR	Call RECUR recursively.
	...	...	...
	BAL	RET,POP	Sequence of instructions for exit; pop up return address and place in REG[PARAM].
	A	POINT,FOURN	Move stack pointer below parameters for this call. FOURN contains 4·N.
	BR	PARAM	Exit to address in REG[PARAM].

the first parameter at the top and the last parameter $N - 1$ words below the top. The next action is the push-down of the return address. The PDP-11 utilizes a special form of the JSR instruction to save the return address automatically, whereas the System/360-370 must explicitly save the return address as the first action of RECUR. The special form of the subroutine call is JSR P,RECUR. The JSR pushes the contents of the linkage register, the P register in this case, onto the stack. Since the P register contains the address of the instruction following the JSR, the return address is automatically saved when RECUR is entered. After the return address is saved in the stack, the ith parameter can be accessed by indexing relative to the stack pointer. When we call the subroutine recursively, we place N new parameters on top of the stack, without destroying the previous parameters. The recursive call is a subroutine call to RECUR, at which point the new return address is placed on the stack. In Example 10.5, at the exit point we first remove the return address from the stack, and then we eliminate the N parameters from the stack by incrementing the stack pointer. The state of the stack after a return from RECUR is identical to the state of the stack just prior to a call on RECUR. Example 10.6 computes the recursive function of Example 10.3.

It is interesting to see how several of the concepts introduced in this chapter and in previous chapters are brought together in Example 10.5. These concepts form a basic set of tools for the construction of intricate and rather powerful computer programs.

Before leaving this section we should consider one final relation of recursive procedures to rooted trees. Compare the rooted trees in Figure 10.2 to the recursive algorithm for postorder traversal in Example 10.4.

EXAMPLE 10.6

The calling program places the value of X on the push-down stack. F returns the value of N! in the AC. The portion of code labeled MULT takes the values in REG[1] and REG[2], performs a multiplication, and leaves the product in REG[1]. POINT is the stack pointer, REG[6].

```
     LABEL    CODE    OPERAND              COMMENTS
1.   POINT    =       %6
2.   AC       =       %0
3.            MOV     THREE,—(POINT)   ; Calling program places
                                        ; value of N in push-
                                        ; down stack.
4.            JSR     %7,F
     ...      ...     ...                  ...
5.   F:       MOV     2(POINT),AC      ; N is under the return
                                        ; address on the stack
                                        ; and is thus the second
                                        ; item from the top. Fetch
                                        ; N from
                                        ; MEMORY[POINT+2].
                                        ; MOV sets the condition
                                        ; code registers N and Z.
6.            BMI     INFINITY         ; If negative, return value
                                        ; of largest positive num-
                                        ; ber.
7.            BEQ     RETURN           ; If N=0 then return
                                        ; value of 1.
8.            DEC     AC               ; Otherwise prepare for
                                        ; recursive call. Form
                                        ; N—1.
9.            MOV     AC,—(POINT)      ; Push down the param-
                                        ; eter.
10.           JSR     %7,F             ; Recursive call on F.
11.           MOV     2(POINT),%1      ; Fetch value of N to
                                        ; REG[1].
12.           MOV     AC,%2            ; Fetch value of F(N—1)
                                        ; from recursive call to
                                        ; REG[2].
13.  MULT:    ...                      ; Multiply N*F(N—1).
14.           MOV     %1,AC            ; Place value of N! in AC.
```

```
     LABEL        CODE  OPERAND            COMMENTS
15.               BR    STACKPOP           ; Prepare to return to
                                           ; calling program.
16.  INFINITY:    MOV   LARGENUM,AC        ; Put largest positive
                                           ; number in AC.
17.               BR    STACKPOP           ; Prepare to return to
                                           ; calling program after re-
                                           ; moving parameter from
                                           ; stack.
18.  RETURN:      MOV   #1,AC              ; Place 1 in AC as value
                                           ; of 0!
19.  STACKPOP:    MOV   (POINT)+@POINT     ; Pop return address
                                           ; from stack, then copy
                                           ; into parameter space
                                           ; at top of stack.
20.               RTS   %7                 ; Return from F.
```

Each tree provides a graphic representation of the activity of the algorithm when the computation is complete. At any point in the computation the node in process is represented by a node in the tree, and the contents of the push-down stack reflect the unique path in the tree from the root node to the current node. Consequently, the maximum stack depth attained during a computation is equal to the maximum level of a leaf in the tree. Every recursive procedure can be represented by a tree like those in Figure 10.2. This is because a predecessor-successor relationship is implied in the recursive definition of a function. In Example 10.3, f(x−1) may be viewed as the successor of f(x). In Example 10.4, f(i.leftlink) and f(i.rightlink) are the successors of f(i).

10.4 STACK INSTRUCTION SETS

In this section we shall examine an entirely different concept in machine instructions, the *stack instruction set.* With such an instruction set we can use postorder representations of binary trees, which in turn represent arithmetic expressions. The postorder representation is, in fact, a program for evaluating an arithmetic expression.

The central notion of the stack instruction set is that for each ordered tree there exists a representation of that tree in a linear string of symbols such that the tree can be reconstructed from its linear representation. For each arithmetic expression we shall construct the linearized form of its tree, and that linearized form is a program for evaluating the expression for a stack machine.

To obtain the linearized representation of an arithmetic expression, we perform the following steps:

1. Represent the arithmetic expression as an ordered rooted tree.
2. Represent the ordered rooted tree as a binary tree, as described in Section 10.2 and Figure 10.6.
3. Traverse the binary tree in postorder. Each time a node is visited, add a symbol for that node to the right end of the linear representation of the tree.

The examples below show the linear representations for typical arithmetic expressions. The ordered rooted tree and binary tree for each expression are shown in Figure 10.8. Our common notation for arithmetic

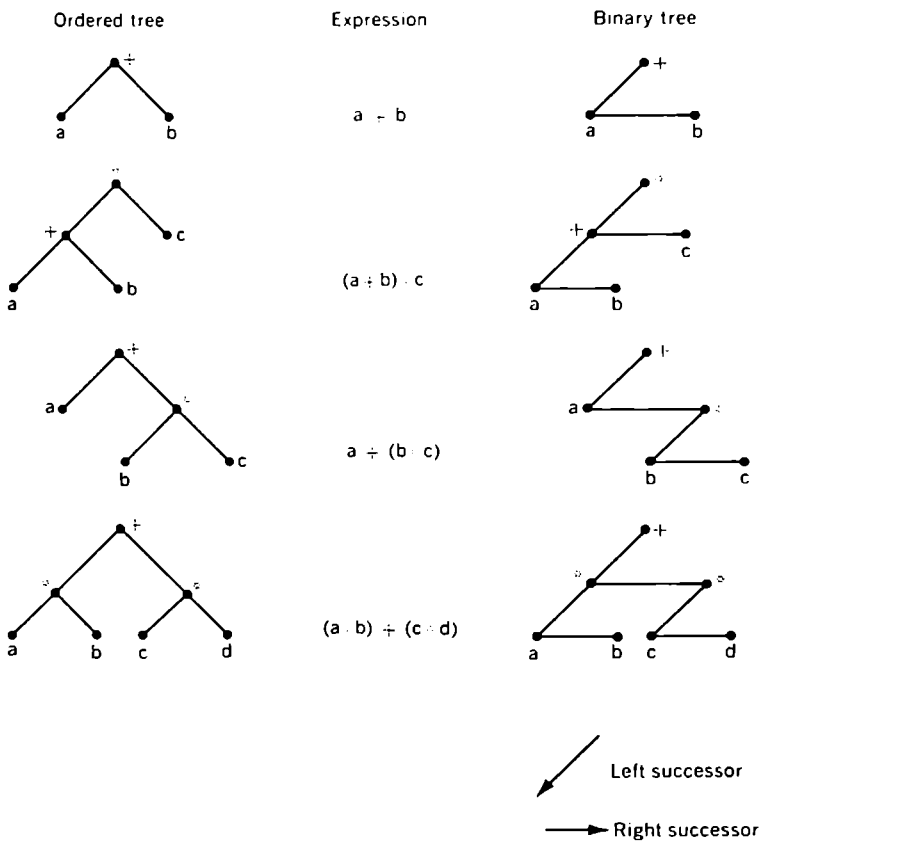


FIGURE 10.8 Ordered trees for arithmetic expressions and their corresponding binary trees

expressions is called *infix notation*, and the linearized form of the tree representation is called *postfix notation* or *Polish notation* (in tribute to the Polish mathematician Jan Łukasiewicz).

Infix	a+b	(a+b)*c	a+(b*c)	a*b+c*d
Postfix	ab+	ab+c*	abc*+	ab*cd*+

The important aspect of postfix notation is that no parentheses are needed to represent arithmetic expressions. Infix notations must resort to parentheses or some other artifice to reconcile the inherent ambiguity in expressions such as $a + b*c$.

All the arithmetic operators that we use in this chapter are binary operators in the sense that each requires two operands. We can represent nodes of arbitrary degree uniquely in Polish notation, provided that we specify the degree of each operator node. For example, if $\oplus$ represents the operator that forms the sum of three quantities, then the expression

$$\frac{a + b}{\oplus(x,y,z)}$$

has the linearized representation

Entity	a	b	$+$	x	y	z	$\oplus$	$/$
Degree	0	0	2	0	0	0	3	2

The nodes of degree 0 are the leaves of the tree.

Let us now see how the linearized representation can be used to reconstruct the ordered-tree representation. Suppose that we are given a linear string of symbols, and that each symbol has a degree associated with it. The following algorithm translates the linear representation into an ordered rooted tree:

1. Starting with the leftmost symbol in the string, repeat step 2 for each symbol in the string, scanning from left to right, and then terminate the algorithm.
2. If the symbol has degree 0, do steps 3 and 4, otherwise do steps 5 to 7.
3. Create a leaf of the tree labeled by the present symbol.
4. Add the symbol to a push-down stack and then terminate the processing of the present symbol.

5. Create a branch node of the tree labeled by this symbol.
6. Let the node have degree n. The n successors of the branch node are obtained by popping up the stack n times and noting the symbols that appear. The nth successor appears first, the $(n-1)$st appears next, and the first successor appears last.
7. Add the present symbol to a push-down stack and terminate the processing of the present symbol.

The unique characterization is the key to the use of linear representations in stack instruction sets. With this in mind, let us postulate a computer with a push-down stack and the following instructions in its instruction repertoire:

VAL	X	The value of X is placed on the top of the push-down stack.
ADD		The top two entries on the stack are replaced by a single item equal to their sum.
MUL		The same as ADD, except the product replaces the top two entries.
DIV		The same as above, except that STACK[TOP—1]/STACK[TOP] replaces the top two items.
SUB		STACK[TOP—1]—STACK[TOP] replaces the top two items.

With this instruction set, the program to compute

$$\frac{ax^2 + bx + c}{dy + e}$$

can be obtained directly from the postfix representation

$$axx**bx*+c+dy*e+/$$

The program is

```
VAL   a
VAL   x
VAL   x
MUL
MUL
VAL   b
```

The correct operation of the stack computer depends on the fact that the computer treats VAL as a degree-0 operator and the other operators as degree-2 operators. Degree-0 operators are executed by stacking an item, and degree-n operators for $n > 0$ are executed by replacing the n items at the top of the stack by a single item. Note how this corresponds to the algorithm for constructing an ordered tree from its linear representation.

The instruction set can be extended to include stack equivalents of all the common operators. Among these are

NAME	X	Load name of X to top of stack.
STO		Store top of stack at address given by STACK[TOP—1] and delete both entries.

Now we can form assignment statements. For example,

$$z := a + b + c*d$$

becomes

```
NAME  z
VAL   a
VAL   b
ADD
VAL   c
VAL   d
MUL
ADD
STO
```

To clarify the operation of the machine, simulate the behavior of the stack machine for this example.

Relational operators and conditional branches make use of the following operators:

EQU	Top two items of the stack are compared for equality. If they are equal, the value **true** replaces the top entries; otherwise the value **false** replaces the top two entries.
LSS	Top two items of the stack are compared numerically. If the next-to-top item is less than the top item, the value **true** replaces the top two entries; otherwise the value **false** replaces the top two entries.

NEQ GEQ LEQ GTR		The other relational operators.
JMP	label	The program branches unconditionally to label, no stack action occurs.
JT	label	Conditional jump if **true** to label. If the value at the top of the stack is **true**, a jump occurs; otherwise no jump occurs. In either case stack is popped once.
JF	label	Conditional jump if **false** to label. Execution similar to JT.

Before we consider the use of these instructions, let us make note of the following instructions for aiding subroutine communication:

CALL	F	Branch to subroutine at location F, store return address at top of stack.
XCH		Exchange top two cells of stack.
RET	n	Return to address at top of stack. Prior to the return pop up stack by n entries. (This deletes n — 1 arguments in stack.)
MARK	x	Store name of current top of stack in location x (used to reference subroutine arguments when top of stack changes).

Now turn to Example 10.7, a program written in the stack instruction repertoire, translated from Example 10.3.

EXAMPLE 10.7

In the instructions below x[y] means that the effective memory address at execution time is determined by computing the value of MEMORY[x]+y.

	LABEL	CODE	OPERAND	COMMENTS
1.	F	VAL	q	Present value of q is saved in stack.
2.		MARK	q	TOP is stored in q. Now q[0] contains the old value of q, q[—1] contains the return address, and q[—2] contains x.
3.		VAL	q[—2]	Bring x to top of stack.

	LABEL	CODE	OPERAND	COMMENTS
4.		VAL	zero	Compare x to 0.
5.		LSS		
6.		JF	next	Jump if $x \geq 0$.
7.		NAME	result	Fetch the name of result.
8.		VAL	infinity	Fetch infinity.
9.		STO		Store infinity into result and exit.
10.		JMP	exit	
11.	next	VAL	q[−2]	Fetch x.
12.		VAL	zero	Fetch 0 and compare to x.
13.		EQU		
14.		JF	recur	Jump if $x \neq 0$.
15.		NAME	result	Set result to 1.
16.		VAL	one	
17.		STO		
18.		JMP	exit	Jump to exit.
19.	recur	NAME	result	Compute result = x*F(x−1); place name of result on stack.
20.		VAL	q[−2]	Place x on stack.
21.		VAL	q[−2]	Compute x − 1 and pass it to F.
22.		VAL	one	
23.		SUB		
24.		CALL	F	After this call, the top three cells contain F(x−1), x, and name of result.
25.		MUL		Compute x*F(x−1).
26.		STO		Store in result.
27.	exit	NAME	q[−2]	Leave the result in the cell now occupied by x, it will be left at top of stack when we exit.
28.		VAL	result	
29.		STO		
30.		NAME	q	Restore value of q.
31.		XCH		Before this instruction is executed stack contains name of q followed by old value of q.
32.		STO		
33.		RET	1	Return address is popped off top of stack; new top of stack contains result.

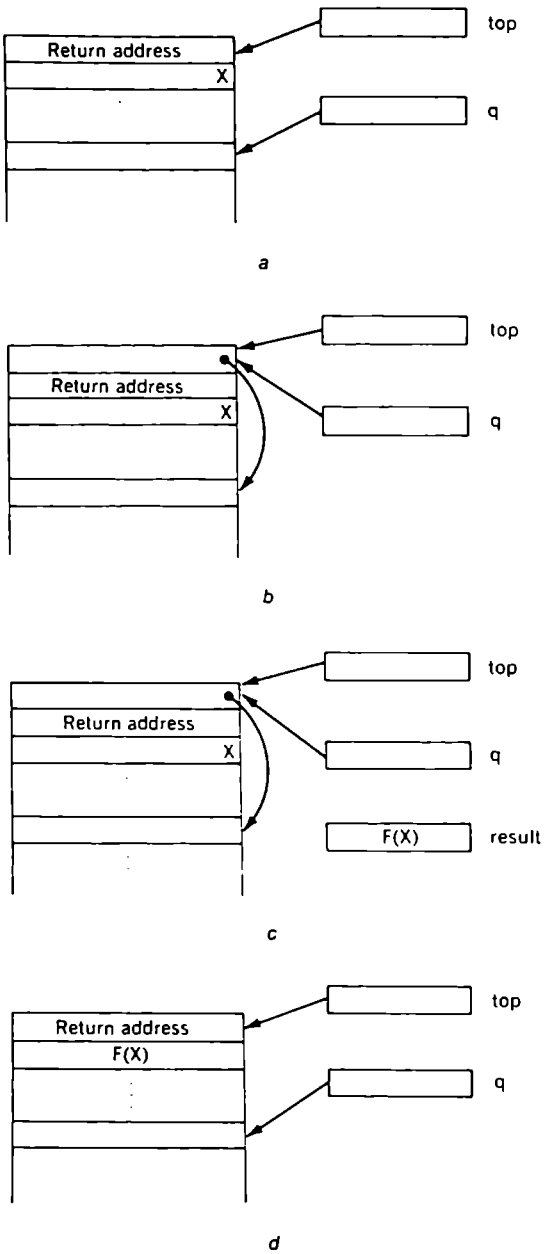


FIGURE 10.9 The state of the push-down stack during the execution of Example 10.6; (*a*) at entry to subroutine, (*b*) after first two instructions, (c) at exit label, and (*d*) before RET is executed

TOP is the current value of the stack pointer; it points to the last item entered in the stack. We use the value of the stack pointer at the entry point of the subroutine as a reference point, q, to access temporary variables and parameters in the stack. Since recursive calls ordinarily destroy this reference point, the example shows how to save the reference-point value by storing it in the stack when a subroutine is entered and restoring it from the stack at the subroutine exit.

At the entry of the subroutine the stack is assumed to have the return address at the top of the stack and the value of the parameter at the next to the top of the stack. The first two instructions store the stack reference pointer at the top of the stack and initialize it to indicate the new top of the stack. Figure 10.9 shows the condition of the stack just before entry into this subroutine, after the execution of the first two instructions, when the instruction labeled exit is reached and when the RET instruction is reached. Note how q, the stack reference pointer, is saved and restored. Note also how the parameter x is accessed in relation to q.

Several machines have been constructed for stack instruction repertoires, although their number is still insignificant. The most prevalent stack machine in the United States is the Burroughs B-5500 computer.

REFERENCES

The originators of the threaded binary tree and the methods for traversing binary trees are Perlis and Thornton (1960). Knuth (1968) provides a thorough and entertaining discussion of the subject which should not be missed by the serious computer scientist. The importance of trees in data structures cannot be attributed to any one application, or even to just a few applications. McCarthy's programming language LISP, invented circa 1959, uses trees as the principal data structure in the language, although trees had been much used in computation for many years before the invention of LISP. The binary search described in the next chapter is based on a tree structure and is also one of the earliest digital-computer algorithms.

The concept of parenthesis-free notation, originated by Łukasiewicz (cf. Łukasiewicz, 1951), has had an important impact on the development of compilers for high-level languages. The notion of the stack machine is a direct outgrowth of parenthesis-free notation. Barton (1961) described a computer with a stack instruction set that was eventually produced as the Burroughs B-5000 and B-5500. The KDF9 computer was the British version of the stack machine and preceded the B-5500

by a few years (Haley, 1962). Current trends in computer hardware indicate that push-down stacks are becoming common, primarily for storing return addresses and parameters for subroutine calls and for saving the status of a processor when an interrupt occurs. However, the stack instruction repertoire for arithmetic processing is still a rarity.

EXERCISES

10.1 Give a nonrecursive algorithm for traversing a binary tree in preorder.

10.2 Give a nonrecursive algorithm for traversing a binary tree in endorder.

10.3 Threaded links can replace empty links in a binary tree without the need for additional memory space. Give two algorithms for traversing a threaded binary tree in preorder. For the first algorithm assume that the threaded links point to the preorder predecessor or successor, and for the second algorithm assume that the threaded links point to the postorder predecessor and successor.

10.4 Convert the following expression into Polish notation:

$$A + B \cdot C \cdot (D + E + F) \cdot G/(H \cdot I \cdot (J + K)) + L/M$$

adding parentheses so that the expression is evaluated from left to right wherever a choice exists. That is, show $X + Y + Z$ as $((X + Y) + Z)$, and not as $(X + (Y + Z))$.

a. Show the ordered-tree representation,

b. Show the binary-tree representation of the ordered tree.

c. Show the linearized form.

10.5 Prove that the leaves of a binary tree are always visited in the same order whether the binary tree is traversed in preorder, postorder, or endorder.

10.6 Write a recursive program for the stack machine that computes the function

$$G(x) = \begin{cases} G(x-1) + G(x-2) & \text{if } x \geq 3 \\ 1 & \text{if } x = 2 \\ 0 & \text{if } x = 1 \end{cases}$$

ELEVEN

SORTING AND SEARCHING

Earlier chapters have been devoted to fundamental aspects of computers rather than to their application to the execution of algorithms. In this chapter we shall discuss several different algorithms for sorting and searching. A rather interesting aspect of algorithms is that the strategies are very different from those that might be employed by a human being doing the same task. The intent of this chapter is to introduce techniques that depend heavily on the capabilities and characteristics of computers.

We shall also consider the analysis of algorithms and the use of some basic combinatorial tools for measuring their relative efficiency.

11.1 SORTING

In this section we consider two different techniques for sorting an array of numbers into numerical order. One, the *bubble sort*, is essentially a "quick-and-dirty" technique. The second, *Floyd's tree sort*, is an elegant method of high efficiency. An interesting result of analysis is that the bubble sort can sort N items in a time proportional to N^2, while the tree sort takes a time proportional to $N \log_2 N$. Thus for sufficiently large N the tree sort is guaranteed to be faster than the bubble sort, regardless of the constants of proportionality.

11.1.1 The Bubble Sort

The name *bubble sort* refers to the characteristic that small numbers "float" to the top of an array if they are placed in the bottom. The bubble sort is very simple to implement, which accounts for its popularity. In applications that require the sorting of very large files, however, its low efficiency makes it unacceptable for use.

The bubble-sort algorithm can be described as follows:

1. Set a flag to 0.
2. Scan the data array from one end to another.
3. Examine pairs of numbers in adjacent words, and if the number with the higher index is numerically smaller, exchange the numbers and set the flag to 1.
4. When the scan is completed, examine the flag. If it is not 0, return to step 1 and repeat the process, otherwise exit.

The claim has been made that this algorithm is inefficient. There remains the problem of justifying this claim. To obtain a machine-independent comparison it is necessary to avoid measures that involve the number of instructions executed or the number of memory accesses. Such measures can vary widely among different computer organizations because they depend strongly on the number of processor registers and on the addressing mechanism. When we analyze an algorithm, we use measures that are not biased by machine organization. Consequently, for the bubble sort we choose to count the number of times the comparison and conditional exchange are executed in the central loop of the algorithm.

The analysis is somewhat complicated by the fact that the number of data exchanges is a function of the initial arrangement of the data.

Therefore, it is necessary to perform an analysis that takes into account the various possible arrangements of data. A natural analysis is a worst-case analysis. For this case we find the maximum number of exchanges that can occur. The analysis of the average case is statistical in nature and involves the calculation of the mean number of exchanges. Since the statistical analysis depends on the initial distribution of data, the usual assumption is that all possible arrangements of data are equally likely. In some instances, however, this assumption is not valid, because it may be the case that the data are "almost" sorted with high probability. Thus our statistical analysis may yield results that are inapplicable to specific situations.

In the simulation of the bubble-sort algorithm given below the values in the array are shown at the beginning of each loop and at the procedure exit. Note that no exchanges are ever made in the last scan.

ARRAY INDEX	PASS 1	PASS 2	PASS 3	EXIT
1	6	3	1	1
2	3	1	3	3
3	1	6	6	6
4	17	17	17	17
5	18	18	18	18
6	26	23	22	22
7	23	22	23	23
8	22	26	26	26

Although we shall analyze only the worst case and the average case here, obviously other analyses may be of interest. For example, we may wish to know the variance of the number of exchanges, as well as higher-order statistical functions. For our purposes the worst-case and average-case analyses are sufficient to illuminate the differences in algorithms.

The worst-case analysis of Example 11.1 is rather simple because of the following observation. If the smallest element in the array happens to be in A[N], then it is moved toward A[1] by one position in each pass. After $N - 1$ passes through the array it is in place, and therefore, counting the last pass, N scans of the entire array might be required, thus bounding the number of comparisons and exchanges by $N \cdot (N - 1)$.

The analysis of the mean number of exchanges is messy by comparison. Nevertheless, it is instructive because it illustrates important techniques. Let us assume that the initial values of A are $a_1, a_2, \ldots, a_N$. Given the initial values, we can compute the set of integers $r_1, r_2, \ldots, r_N$,

EXAMPLE 11.1

A version of the bubble-sort algorithm in ALGOL is as follows:

```
procedure sort (A,N);
array A;
integer N; comment lower bound of A is 1 and upper bound is N;
begin integer i; boolean flag; real temp;
    loop:
    flag := false;
    for i := 1 step 1 until N-1 do
        begin
            if A[i] > A[i+1] then
            begin comment interchange;
                temp := A[i];
                A[i] := A[i+1];
                A[i+1] := temp;
                flag := true;
            end;
        end;
    comment another iteration is made if flag is true;
    if flag then go to loop;
comment the array is now sorted in ascending numerical order;
end sort;
```

where r_i is defined to be the number of elements that precede a_i in the sequence and that are greater than a_i. We call r_i the *ranking* of a_i. For the sequence of numbers 2, 1, 4, 3, the corresponding sequence of rankings is 0, 1, 0, 1. It should be clear that the number of exchanges depends only on the sequence of rankings. That is, if two initial conditions of the A array are associated with the same ranking sequence, then both initial conditions require the same number of exchanges when they are sorted.

Simple combinatorial theory tells us that there are $N!$ possible rearrangements of a sequence of N numbers. For each of these rearrangements there is a ranking sequence. It is rather interesting to observe that each of these ranking sequences is distinct. Moreover, for each ranking sequence there is a unique permutation of the N numbers. To see this, consider the different values that each r_i can assume. Since no element precedes a_1, r_1 must be 0. Since one element precedes a_2, r_2 may be either 0 or 1. In general $i-1$ elements precede a_i, so that $0 \leq r_i \leq i-1$. Then r_i can assume i different values. Thus there are no more than $1 \cdot 2 \cdot 3 \cdots N = N!$ ranking sequences of length N, and for each ranking sequence we can construct one of the $N!$ distinct permutations of N numbers. For example, suppose that the ranking sequence is 0, 0, 2, 1.

Which permutation of the integers 1 to 4 is described by this ranking sequence? The last integer in the sequence must be 3, because its rank is 1. The third integer must therefore be 1, because it has a rank of 2 with respect to integers 1, 2, and 4. From this line of reasoning the permutation 2, 4, 1, 3 is the unique permutation of the integers 1 to 4 that has the ranking sequence 0, 0, 2, 1. An algorithm for constructing permutations from ranking sequences is the subject of Exercise 11.1.

The notion of ranking sequence permits us to calculate the average number of times that the label loop is passed, which is the number of executions of the main loop of the program. Note that if a particular element has rank m in a permutation, then the main loop will be repeated at least $m + 1$ times during the sort. In fact we see that the main loop is repeated $k + 1$ times when the maximum rank of an element in the A array is equal to k. The average number of iterations of the main loop is therefore given by

$$\text{Average iterations} = \sum_{i=1}^{N} i \cdot \Pr(\text{max rank} = i - 1)$$

where $\Pr(\text{max rank} = i - 1)$ is the probability that the maximum ranking of a permutation is exactly $i - 1$. Following the same notation, we let

$$\Pr(\text{max rank} \geq i - 1) = \sum_{j=i-1}^{N-1} \Pr(\text{max rank} = j)$$

Then the summation for the average number of iterations becomes

$$\begin{aligned}
\text{Average iterations} &= \sum_{i=1}^{N} i \cdot \Pr(\text{max rank} = i - 1) \\
&= \sum_{i=1}^{N} \sum_{j=1}^{i} \Pr(\text{max rank} = i - 1) \\
&= \sum_{j=1}^{N} \sum_{i=j}^{N} \Pr(\text{max rank} = i - 1) \\
&= \sum_{j=1}^{N} \Pr(\text{max rank} \geq j - 1) \\
&= \sum_{i=1}^{N} [1 - \Pr(\text{max rank} < i - 1)] \\
&= \sum_{i=0}^{N-1} [1 - \Pr(\text{max rank} < i)]
\end{aligned}$$

Note the limits on the summations in the second and third lines above. It is easily verified that the double summations treat the same pairs of indices i and j. If we assume that all $N!$ permutations of N integers are equally likely, then the term Pr(max rank $< i$) has a simple form.

To find Pr(max rank $< i$) we shall count the number of ranking sequences that have no ranking with a value i or greater. Since the first i rankings of a ranking sequence cannot exceed $i - 1$, we can select the first i rankings in $i!$ different ways. For j in the interval $i + 1 \leq j \leq N$ each r_j can assume any integer value in the interval $0 \leq r_j \leq i - 1$. Thus for each of the last $N - i$ rankings we have i choices of values for the rankings. Therefore the total number of ranking sequences with rankings that do not exceed i is $i! \cdot i^{N-i}$. From this we find

$$\text{Pr(max rank} < i) = \frac{i!i^{N-i}}{N!}$$

and

$$\begin{aligned}\text{Average iterations} &= \sum_{i=0}^{N-1}\left[1 - \frac{i!i^{N-i}}{N!}\right] \\ &= N - \frac{1}{N!}\sum_{i=0}^{N-1} i!i^{N-i}\end{aligned}$$

The summation term in this formula is rather messy to evaluate. Although it does not lend itself to simplification, Knuth (1968) has found that it approaches

$$\sqrt{\frac{\pi N}{2}} - \frac{5}{3} + \frac{11}{24}\sqrt{\frac{\pi}{2N}} + \cdots$$

asymptotically as N increases. The terms left out of this expansion contain N^{-1} or higher negative powers of N and are therefore dominated by the leading terms of the expansion for large N. Thus we find that the average number of iterations is given by

$$\begin{aligned}\text{Average iterations} &= N - \sqrt{\frac{\pi N}{2}} + \cdots \\ &\cong N - 1.25\sqrt{N}\end{aligned}$$

Since each iteration involves $N - 1$ comparisons, the average number of comparisons is approximately

$$(N - 1)(N - 1.25\sqrt{N}) = N^2 - 1.25N^{3/2} - N + 1.25\sqrt{N}$$

which grows asymptotically like the maximum number of comparisons, $N^2 - N$, as N becomes large. This suggests that it may be more efficient to eliminate the setting and testing of the flag in Example 11.1 and merely iterate the main loop $N - 1$ times. The flag is supposed to help us eliminate unnecessary comparisons, but our analysis indicates that the work expended in maintaining the flag may exceed the work that is saved.

In order to improve the bubble sort we can use a strategy that moves the small numbers toward A[1] as rapidly as the large numbers are moved toward A[N]. This yields Example 11.2.

EXAMPLE 11.2

```
for j := 1 step 1 until N−1 do
if A[j+1] < A[j] then
    begin
    temp := A[j+1];
    for i := j step −1 until 1 do
        begin if temp ≥ A[i] then go to endiloop;
            A[i+1] := A[i];
        end;
    i := 0;
    endiloop: A[i+1] := temp;
    end;
```

When the algorithm of Example 11.2 is executed on the array A in Example 11.1, array A attains the following states, shown at the statement

```
temp := A[j+1];
```

ARRAY INDEX	j = 1	j = 2	j = 6	j = 7	FINISH
1	6	3	1	1	1
2	3	6	3	3	3
3	1	1	6	6	6
4	17	17	17	17	17
5	18	18	18	18	18
6	26	26	26	23	22
7	23	23	23	26	23
8	22	22	22	22	26

In the inner loop of the algorithm, if a small A[j+1] is encountered, then it is moved toward A[1] as far as possible. The outer loop of the algorithm increases j, so that each inner loop treats an array element that has never been scanned before. The branch to the label endiloop is taken when the proper place has been found for the element A[j+1].

The analysis for this algorithm requires only a minor variation of the analysis of Example 11.1. Clearly in the worst case the interchange will always occur. In this case the element that occupies A[j+1] initially will move j places in the inner loop. Then the maximum number of comparisons in the inner loop is given by

$$\text{Maximum comparisons} = \sum_{j=1}^{N-1} j = \frac{N(N-1)}{2}$$

This is exactly half the comparisons for the worst case of Example 11.1.

To find the average number of comparisons we note that if all permutations are equally likely, then all the possible values of the ranking r_j are all equally likely. Moreover, the values of r_j are independent of the values of every other ranking in a sequence. During the jth iteration of the outer loop of the algorithm A[j+1] will be exchanged r_{j+1} times and will be compared $r_{j+1} + 1$ times. The average total number of comparisons is given by a summation of the average number of comparisons for each A[j], where $2 \leq j \leq N$. That is, if we let the average value of r_j be denoted by $\hat{r}_j$, then

$$\text{Average comparisons} = \sum_{j=2}^{N} \hat{r}_j + 1$$

However, it is equally likely that r_j is any integer in the interval $0 \leq r_j \leq j - 1$. Thus

$$\hat{r}_j = \frac{1}{j} \sum_{i=0}^{j-1} i = \frac{j-1}{2}$$

Substituting for $\hat{r}_j$ in the above formula yields

$$\text{Average comparisons} = \sum_{j=2}^{N} \frac{j+1}{2} = \frac{(N+4)(N-1)}{4}$$

In comparing the two versions of the bubble-sort algorithm we see

that Example 11.1 requires an average number of comparisons that is dominated by N^2, whereas Example 11.2 requires an average that is dominated by $N^2/4$. Thus for sufficiently large N the second bubble-sort algorithm is likely to be 4 times as fast as the first when all permutations of the data are equally likely.

The analyses in this section are typical. Worst-case analyses are a crude, but often revealing, basis for comparing algorithms. Analysis of the average case is generally more accurate, but it is more difficult. In our examples above the N^2 term was dominant in both the average-case and worst-case analyses, although the multiplicative constant for the N^2 term differed. For some algorithms the dominant term for the worst case can differ from that for the average case by more than just a constant coefficient.

Although the analysis in this section indicates that the algorithm of Example 11.2 is better than that of Example 11.1, the next section shows that Example 11.2 is actually not a very good algorithm at all. We shall see that it is possible to sort N numbers with a number of comparisons proportional to $N \log_2 N$, which grows much more slowly with N than does N^2.

11.1.2 Tree Sort

A very interesting sorting algorithm was reported by Floyd (1964). The strategy in this case is to organize the data in the array into a tree in the first phase and then rearrange the tree into a sorted list in the second phase. Both phases can be executed efficiently, which leads to a very efficient algorithm.

Figure 11.1 shows an example of a tree in the form used in Floyd's algorithm. Each node in the graph represents an element of a data array, and the index of the element associated with each node appears as a node label. Each interior node of the tree has one node above it in the tree—a predecessor node—and one or two nodes below it in the tree—successor nodes. The predecessor of A[i] is A[i÷2] (recall that ÷ denotes integer division), and the successors of A[i] are A[2*i] and A[2*i+1].

In the first phase of the algorithm a tree is so constructed that each node is numerically greater than or equal to its successors; that is, $A[i] \geq A[2*i]$ and $A[i] \geq A[2*i+1]$ for all i. This is done by adding new nodes to the tree one at a time. As each node is added, it is compared to its predecessor and exchanged with it if necessary. If an exchange occurs, then the new item is compared with the predecessor of its current position to see whether another exchange must be made. The process of comparing and exchanging continues until a place is found for the new

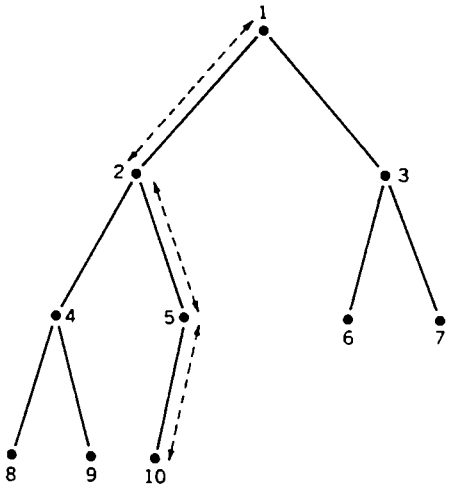


FIGURE 11.1 Floyd's tree. Dashed lines show comparison that might be made when tenth item is added to tree

item at a node within the tree such that it is less than its predecessor. The number of exchanges required to insert one item is bounded by the number of levels in the tree rather than by the number of items in the tree. Thus at most $\lfloor \log_2 N \rfloor$ comparisons and exchanges are needed to find the position of a new item when there are N items in the tree ($\lfloor x \rfloor$ is the largest integer not exceeding x).

Figure 11.1 illustrates a case in which the tenth item is to be placed in the tree. First A[10] is compared with its predecessor, A[5]. If A[10] is greater than A[5], then A[5] will be exchanged with A[10], and the new A[5] will be compared to A[2]. If an exchange occurs after this comparison, the new A[2] will be compared to A[1], and an exchange will take place if necessary.

Once the data are arranged in the form of a tree, they can be sorted by comparing and exchanging from the top down rather than from the bottom up. The largest element in the array is in A[1], so that it can be moved to A[N]. When it is moved to A[N], a new Floyd tree is formed from the remaining $N - 1$ elements of the tree. The larger of A[2] and A[3] moves into A[1], and the largest of its successors moves up one notch in the tree. This process terminates when we find a place for the element displaced from A[N]. The result of these shifts in the tree is that the largest number among the $N - 1$ in the new subtree is found in A[1]. The iteration is repeated by exchanging A[1] and A[N−1] and forming a new tree with $N - 2$ elements. The entire array is sorted by finding the largest element among the first i elements of A and placing this element in position A[i], where the index i runs backward from N to 2.

EXAMPLE 11.3

The following version of Floyd's tree sort is drawn partly from tree sort and partly from an algorithm called *heap sort*, which stimulated tree sort and is due to Williams (1964).

```
procedure treesort(A,N);
integer N; array A;
comment the array A is declared [1:N];
    begin
    procedure intree(lim,in);
    integer lim; real in;
    comment procedure for first-phase processing.
    The variable in is the new datum to be placed in the tree;
    begin
        i := lim;
        scan: if i > 1 then
            begin j := i ÷ 2;
                if in > A[j] then
                    begin A[i] := A[j];
                        i := j;
                        go to scan;
                        comment part of an exchange. The variable in is not
                        actually placed into the tree until just prior to exit;
                    end;
            end;
        A[i] := in;
    end procedure intree;
procedure outtree(lim);
integer lim;
comment this procedure finds the largest of the first lim elements in the tree
and places that element in A[1] by exchanging elements along a tree path;
    begin real copy; integer i,j;
        i := 1;
        copy := A[1];
    loop: j := 2*i;
        if j ≤ lim then
            begin if j < lim then
                begin comment two successors;
                if A[j+1] > A[j] then j := j+1;
                comment select the largest successor;
                end;
            if A[j] > copy then
                begin A[i] := A[j];
                    i := j;
                    comment the larger element is moved up the tree;
                    go to loop;
```

```
            end;
        end;
    A[i] := copy;
end procedure outtree;
comment the body of procedure treesort begins here;
integer i; real temp;
    for i := 2 step 1 until N do
        intree(i,A[i]);
    for i := N step -1 until 2 do
        begin
            temp := A[1];
            A[1] := A[i];
            A[i] := temp;
            comment largest element in tree is placed in A[i];
            outtree(i-1);
        end;
end procedure treesort;
```

Below are some of the intermediate states of the array A as it is sorted by the tree-sort algorithm of Example 11.3. The state of the array is shown at the exit points of the intree and outtree procedures.

ARRAY INDEX	INITIAL VALUE	INTREE i = 4	INTREE i = 8	OUTTREE i = 7	OUTTREE i = 4	FINAL VALUE
1	6	17	26	22	6	1
2	3	6	22	17	3	3
3	1	1	23	18	1	6
4	17	3	17	3	17	17
5	18	18	6	6	18	18
6	26	26	1	1	22	22
7	23	23	18	23	23	23
8	22	22	3	26	26	26

A cursory examination of Example 11.3 reveals that procedures intree and outtree are very similar. Indeed, Floyd's algorithm is structured so that a single procedure performs both tasks. The Williams algorithm has been presented because the distinction in the procedures adds to the clarity.

The worst-case analysis for the tree sort can be done by assuming that an exchange follows every comparison. We shall count the number of times that the labels scan and loop are encountered in the worst case and use this as a measure of the computational work.

In the worst case an element in a tree of size i will cause loop or scan to be passed $\lfloor \log_2 i \rfloor$ times. Since we deal with a tree of size i for each i in

the interval $2 \leq i \leq N$, the maximum number of iterations through either scan or loop is given by

$$\begin{aligned}\text{Maximum iterations} &= \sum_{i=2}^{N} \lfloor \log_2 i \rfloor \\ &= 1 + 1 + 2 + 2 + 2 + 2 + 3 + 3 + \cdots \\ &\cong \sum_{j=1}^{\lfloor \log_2 N \rfloor} j2^j\end{aligned}$$

The last approximation is an equality when N is one less than a power of 2. It is particularly convenient to use this form of the summation, because it can be rewritten in a simple closed form. To derive the closed form we note that the term $j2^j$ is equal to the derivative of the term $2x^j$ when x has the value 2. Consequently we can employ the polynomial

$$g(x) = 1 + x + x^2 + \cdots + x^n = \frac{x^{n+1} - 1}{x - 1}$$

where we let $n = \lfloor \log_2 N \rfloor$. Then the sum we desire is equal to $2g'(x)$ evaluated at $x = 2$. When we take the derivative of $(x^{n+1} - 1)/(x - 1)$, we find

$$\begin{aligned}\text{Maximum iterations} &= 2\, g'(2) \\ &= \frac{2(n+1)x^n(x-1) - 2x^{n+1} + 2}{(x-1)^2}\Bigg|_{x=2} \\ &= (n-1) \cdot 2^{n+1} + 2\end{aligned}$$

When we replace n by $\lfloor \log_2 N \rfloor$ and let N be one less than a power of 2, the formula has the form

$$\text{Maximum iterations} = (\lfloor \log_2 N \rfloor - 1)(N + 1) + 2$$

The general conclusion that we reach here is that the amount of computation in the worst case is asymptotically proportional to $N \log_2 N$. This grows much more slowly with N than does N^2, the function that describes the worst-case computation for the bubble sort. Actually $N \log_2 N$ grows more slowly than $N^{1+\epsilon}$ for any positive ϵ. Thus the worst case for the tree sort will require less computation than the worst case for the bubble sort, provided that N is sufficiently large.

The analysis of the average computation for the tree sort is still an open problem. Nevertheless, there is an information theoretical argument that can be used to provide a lower bound for the average computation. Suppose that on the average $Q(N)$ comparisons are required to sort N numbers. Since each comparison has two possible outcomes, there are essentially $2^{Q(N)}$ different outcomes for the set of $Q(N)$ comparisons. Surely each of the $N!$ permutations of N items must result in a different set of outcomes for the $Q(N)$ comparisons. If not, two different permutations would be treated identically by the sorting algorithm, and the algorithm could not sort both of them. Since each outcome is distinct, we have

$$2^{Q(N)} \geq N!$$

By taking logarithms of both sides and using the Stirling approximation,

$$\log_2(N!) = N \log_2 N - N \log_2 e + \cdots$$

we obtain

$$Q(N) \geq N \log_2 N - N \log_2 e + \cdots$$

Therefore, the average number of comparisons grows asymptotically at least as fast as $N \log_2 N$. Since the worst case for the tree-sort algorithm has a computational complexity that grows as $N \log_2 N$, tree sort is at least within a multiplicative constant of being an optimal sorting algorithm.

Our analyses have essentially established that tree sort is a better sorting algorithm than either of the bubble-sort algorithms that we investigated. However, there are several assumptions embedded in the analyses that can be violated in particular cases. It is crucial, for example, that the approximations are asymptotic. When N is small, the asymptotic approximations can lead to erroneous conclusions. It is also crucial that we compute the average of comparisons under the assumption that all permutations are equally likely. If, by chance, data are presented to the sort algorithm only "slightly" disordered, then the bubble sort will be more efficient than our analysis suggests. Moreover, the tree-sort algorithm tends to require higher computation for data that are "almost" sorted than for other data (see Exercise 11.2). Thus there are peculiar cases for which the apparently inefficient bubble sort out-

performs the very efficient tree sort. The selection of a sorting algorithm ultimately depends on its behavior in the context in which it is used.

11.2 SEARCH ALGORITHMS

In many different applications it is necessary to search through an array or data structure for a particular datum. For example, in the second pass of a two-pass assembly each reference to a symbol causes the assembler to search a table for an entry containing that symbol. When such an entry is found, other information associated with the symbol in the table is used by the assembler to generate a fragment of an assembled word. This section describes three important and widely used techniques for searching tables.

Novice programmers nearly always write search programs that are similar to the following example:

EXAMPLE 11.4

```
for i := 1 step 1 until N do
    begin if table[i] = searchkey then go to found;
    end;
    go to notfound;
found: whereitsat := i;
    . . .
notfound: . . .
```

Obviously, in the worst case N comparisons are required, and the average number of comparisons is $N/2$. Searches of this type should be avoided in programs where the size of the table searched or the frequency of use of the program places a great deal of importance on running efficiency. Better methods for searching are treated in the sections that follow.

11.2.1 The Binary Search

A conceptually simple search technique can be used if the data are sorted numerically in the search table. This technique, called *binary search*, consists of a sequence of iterations such that the area of the table to be searched is halved in each iteration. Eventually either the item is found, or the area to be searched becomes empty. In the latter case the program exits and notifies the calling program that the item is not in the table.

EXAMPLE 11.5

A procedure for the binary search is as follows:

```
procedure BINARY(A,N,FOUND, PLACE, ITEM);
array A; integer N, PLACE, ITEM; boolean FOUND;
comment array A is declared [1:N]. If ITEM is in A, then FOUND is set true
and the index of ITEM in A is stored in PLACE, otherwise FOUND is set false;
begin integer LOW, HIGH;
    LOW := 0;
    HIGH := N + 1;
    FOUND := false;
loop:
    if LOW + 1 ≥ HIGH then go to notfound;
    PLACE := (LOW + HIGH) ÷ 2;
    if ITEM = A[PLACE] then go to foundit
    else if ITEM < A[PLACE] then HIGH := PLACE
    else LOW := PLACE;
    go to loop;
foundit: FOUND := true;
notfound:
end procedure BINARY;
```

In the program above the variable HIGH is initialized to N+1, because PLACE can attain values only less than or equal to HIGH−1. This is because the function (LOW+HIGH)÷2 can equal HIGH only if LOW equals HIGH, but since LOW is initialized to a number less than HIGH, both PLACE and LOW never attain the value of HIGH.

To illustrate the operation of the binary-search algorithm suppose that the array A contains the integers 1 to 6 in A[1] to A[6]. LOW and HIGH are initialized to 0 and 7, respectively, by the algorithm. If ITEM has the value 0, then PLACE would assume the sequence of values 3 and then 1 before a branch was taken to notfound. If ITEM has the value 4, then PLACE would assume the sequence of values 3, 5, and 4 prior to the branch to foundit. The maximum number of iterations performed by the algorithm is $\lfloor \log_2 N \rfloor + 1$ when A contains N elements. The logarithmic behavior of the algorithm is strikingly better than the behavior of the search in Example 11.4.

Figure 11.2 illustrates how the binary search is related to binary trees. The sequence of probes is shown for an array with 15 elements. When the outcome of a comparison shows that ITEM is less than an array element, the next element to be probed is indicated by an arrow slanting

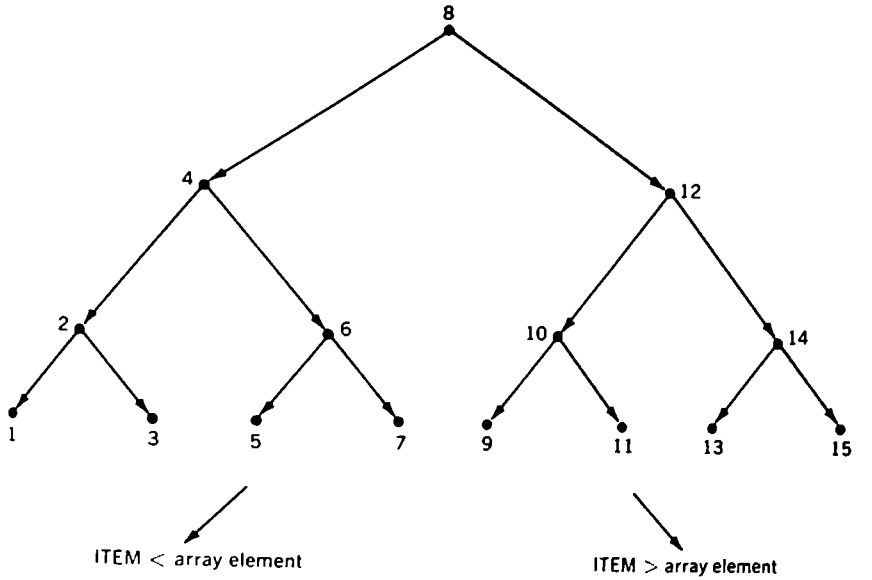


FIGURE 11.2 The sequence of probe addresses in a binary search for various values of ITEM

downward to the left. If ITEM is greater than the array element, then the next probe of the array is indicated by the arrow slanting downward to the right. The tree in Figure 11.2 is clearly a binary tree, but the links to the successors are not stored in the tree as shown in Chapter 10. In this case we find the successors of a node by manipulating HIGH and LOW.

The analysis of the average number of comparisons is similar to the analysis of Floyd's tree-sort algorithm. If an item is not in the array, then either $\lfloor \log_2 N \rfloor$ or $\lfloor \log_2 N \rfloor + 1$ comparisons are required to discover this fact. When N is one less than a power of 2, the average is exactly $\lfloor \log_2 N \rfloor + 1$, because every item that is not in the array will require this many comparisons, regardless of its value.

If an item is in the array, then there is exactly one position in the array that requires one comparison, there are two that require two comparisons, four that require three comparisons, and, in general, 2^j positions that require $j - 1$ comparisons. Thus the average number of comparisons required to find an item in the array is given by

$$\text{Average comparisons} \cong \frac{1}{N} \sum_{j=0}^{\lfloor \log_2 N \rfloor} (j+1)2^j \cong \frac{1}{N}\left(\sum_{j=0}^{\lfloor \log_2 N \rfloor} j2^j + N \right)$$

The formula is exact when N is one less than a power of 2. The summa-

tion was given in the previous section. Recalling that it is equal to $(\lfloor \log_2 N \rfloor - 1)(N + 1) + 2$, we find

$$\text{Average comparisons} = \frac{\lfloor \log_2 N \rfloor (N + 1) + 2}{N}$$

which is approximately $\lfloor \log_2 N \rfloor$ for large N.

Thus the average number of comparisons is approximately one less than the maximum number of comparisons when the item is not in the array. Note that it does not make much difference whether or not the item is in the array. The comparisons required for the two cases differ in number by a small constant. For the sequential search of Example 11.4, it makes a great deal of difference whether or not an item is in the array. In fact, on the average, twice as much computation is required when an item is not in the array as when it is in the array.

11.2.2 Hash Addressing

The amount of computation required for a binary search is a function only of the number of elements in the array to be searched. For some applications, particularly with large arrays, it is desirable to be able to search faster than is possible with a binary search. There is another search algorithm with the interesting characteristic that the amount of computation required depends both on the size of the array to be searched and on a second parameter which can be selected by the programmer. Thus the programmer can control the amount of computation required for a search by setting the value of the second parameter. The choice of the second parameter reflects a trade off between computation speed and memory requirements.

This search algorithm, commonly called *hash addressing*, is based on the idea that the item that is the search parameter can be used as an index to an array. Suppose, for example, that the variable ITEM can assume only the values 1, 2, . . . , M. Then a convenient way of storing search information in an array A is to store a 1 in A[i] if the number i is present in the array and to store a 0 in A[i] otherwise. Now we can easily search A by direct entry to the array. The test

```
if A[ITEM] = 1 then . . .
```

constitutes a search for ITEM in the array. This example points out the use of memory to save computation. M could be very large compared to the number of items that are actually stored in A—say, 10^6 times as

large. Then a binary search would require only a fraction of the storage required by the direct-entry algorithm, but the binary search would require several comparison iterations instead of the single comparison required by direct entry.

When ITEM can assume a very large number of values, but only a very few different possible values are to be stored in A, a *hash code* can be used somewhat like a direct-entry index. Suppose that ITEM is a string of five or fewer eight-bit characters; for example, it might be the internal representation of a symbol from an assembly language program. Suppose that each character has eight bits, so that there are roughly $(2^8)^5 = 2^{40} \cong 10^{12}$ possible symbols, not all of which are valid assembly language symbols. Clearly we cannot use direct entry with such symbols, because the memory requirements are far in excess of available memory. In the typical case only a few hundred items may actually be stored in a list, so that it is possible to use an algorithm similar to direct entry, but with much smaller memory requirements.

Instead of using the value of ITEM as an index, we shall compute a new value by calling a procedure named HASH, with ITEM as an argument. The value of HASH(ITEM) is some number in a range comparable in size to the number of items in the search list. For example, HASH(ITEM) might be a number between 1 and 256. We then use HASH(ITEM) as the index to A, as in the direct-entry algorithm. That is, we use the test

```
if A[HASH(ITEM)] = 1 then . . .
```

There is a problem, however. Two or more different values of ITEM may yield the same hash code, but there may be room to store only one of the items in A[HASH(ITEM)]. Thus special steps must be taken to handle the case when two or more entries have the same hash code. We solve this problem as follows. The hash code effectively partitions the possible values of ITEM into disjoint sets such that if there are N different hash codes, then there are N disjoint sets of symbols with the same hash code. Given a list to be searched, we can divide that list into N sublists according to hash code. Then, to see whether a given value of ITEM is in the list, we first compute its hash code and then search the sublist of items that have the same hash code. The search should involve only $1/N$ of the entire list, on the average, if the function HASH is well constructed.

Example 11.6 illustrates how to search a list by hash addressing and how to enter items into a list in preparation for this search. We assume that the hash codes are the integers $1, 2, \ldots, N$. The array TABLE contains the items. Items with identical hash codes are linked together as one-way linked lists, with the links for these lists stored in NEXT. Thus the item that follows TABLE[i] on a list is TABLE[NEXT[i]]. The beginning

of each list of items with the same hash code is identified by the array INDEX. That is, INDEX[h] contains the index in TABLE of the first item on the list of items with hash code h. We use the value 0 in INDEX[i] if there is no item i in TABLE.

To search TABLE we start at INDEX[HASH(ITEM)] and look for a match. Let this index be called the *index of the first probe* of the table. If there is no match, then we trace a linked list of entries that have the same hash code as ITEM. The next entry on the list is determined by using the index of the first probe as an index to NEXT. The entry in NEXT is the *index of the second probe*, the next item to examine. The link tracing is repeated until either a match is found or a 0 is found in the entry of NEXT. The 0 is the signal that the sublist of entries with the same hash code as ITEM has been exhausted.

The procedure ENTER places items in TABLE. We assume that NEXT, INDEX, and the variable AVAIL have been initialized to 0s before the first call of ENTER. When ENTER is called, it computes the hash code of ITEM and places it at the next available place in TABLE, as indicated by AVAIL. ENTER then modifies an entry in INDEX and in NEXT so that the new item is effectively added to the sublist of entries with a hash code equal to HASH(ITEM).

The worst case would be one in which all items have the same hash code. This case reduces to a search of one list, which is essentially equivalent to the sequential search in Example 11.4. Thus in the worst case we may have to inspect all the items in the array.

EXAMPLE 11.6

```
procedure HASHSEARCH(ITEM, PLACE);
integer ITEM;
begin comment if ITEM is in TABLE, then the index of ITEM is put into
PLACE, otherwise PLACE is set to 0;
    integer i;
        PLACE := 0;
        i := INDEX[HASH(ITEM)];
    loop:
        if i = 0 then go to exit;
        if TABLE[i] ≠ ITEM then
            begin i := NEXT[i];
                go to loop;
            end;
        PLACE := i;
    exit:
```

```
end procedure HASHSEARCH;

procedure ENTER(ITEM);

integer ITEM;

begin integer i;
    i := HASH(ITEM);
    comment the value of INDEX[i] is the index of another entry of TABLE
    with hash code i. Place INDEX[i] in NEXT[AVAIL] so that TABLE
    [INDEX[i]] will be scanned immediately after TABLE[AVAIL] is ex-
    amined during a search;
    AVAIL := AVAIL + 1;
    comment there should be a test to guard against table overflow;
    comment this forces the search to look at TABLE[AVAIL] first;
    TABLE[AVAIL] := ITEM;
    NEXT[AVAIL] := INDEX[i];
    INDEX[i] := AVAIL;

end procedure ENTER;
```

Assume that we call ENTER with the sequence of items below. For this case assume that the hash codes are the integers 0 to 6.

ITEM	HASH CODE
ALPHA	4
BETA	2
GAMMA	5
DELTA	5
EPSILON	0
MU	1
NU	5
OMEGA	2

Then the arrays TABLE, INDEX, and NEXT would have the values given below and illustrated in Figure 11.3.

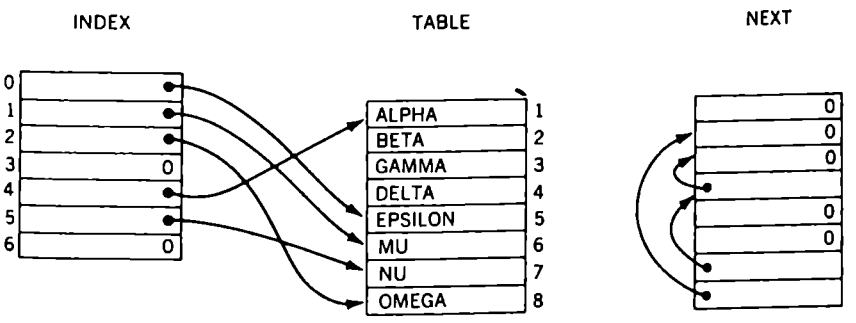


FIGURE 11.3 An example of a hash table for the hash search algorithm

ARRAY ELEMENT	INDEX	TABLE	NEXT
0	5 (EPSILON)	. . .	. . .
1	6 (MU)	ALPHA	0
2	8 (OMEGA)	BETA	0
3	0	GAMMA	0
4	1 (ALPHA)	DELTA	3 (GAMMA)
5	7 (NU)	EPSILON	0
6	0	MU	0
7		NU	4 (DELTA)
8		OMEGA	2 (BETA)

A search for GAMMA will scan NU, and then DELTA, before reaching GAMMA.

The analysis of the average case is straightforward and is given as an exercise. Intuition suggests that, on the average, each sublist will contain $1/N$ of the total number of items, where N is the number of distinct hash codes. Statistical analysis supports this reasoning. On the average, we have to search half of a linked list if an item is in the array, and we must search the entire linked list if the item is not there. Comparison of both of these cases indicates that the hash-addressing algorithm is N times faster than Example 11.4 when the hash codes are well distributed.

The next problem is to determine suitable hash functions. Some experimentation has shown that the function

```
HASH(x) = (x MOD N)+1
```

yields a number uniformly distributed between 1 and N for many collections of data. The distribution is exactly uniform when x ranges over all integers. Since the items usually encountered in searching applications are composed of characters, it is well to consider this in the selection of N. For example, if characters are eight bits, then the choice of $N = 256$ effectively causes the hash code to depend only on the last character in an item. It is safest to select N to be a prime or a number that is not highly composite to avoid the possibility of nonuniform distributions.

Other functions that work well are the logical EXCLUSIVE OR function and ordinary addition. These are usually implemented by breaking an item into several fields of equal size—say, r bits each—and then com-

bining the fields into a single r-bit field with the EXCLUSIVE OR operation or addition. If addition is used, then carries outside the field are disregarded. The result is a hash code between 0 and $2^r - 1$. The reason addition or EXCLUSIVE OR is used is that the final hash code depends on every r-bit field. We should not choose r to be equal to the number of bits per character if items are derived from character strings, because the nonuniform distribution of characters in text will tend to yield a nonuniform distribution of hash codes.

N must be selected to balance search time against memory utilization, since the size of the array INDEX increases in proportion to the size of N. In System/360-370 computers a large memory is available, so that N as large as 1000 might be used to decrease search time in large arrays. However, this is far too large for the PDP-11 computer, which has a very small memory.

The most serious disadvantage of the hash search algorithm is the use of memory for storing linked-list pointers. In Example 11.6 the pointers are stored in the array NEXT. In most implementations of the hash search algorithm the pointers will be part of a packed data structure in order to conserve memory. There is an alternative hash search algorithm that needs no pointer storage at all. Since pointers are necessary only when two or more items have the same code, we can dispense with them by developing a different method for dealing with hash-code duplications. One method that has been widely used is the following:

1. Use a hash code to determine the address of the first probe of a table.
2. If the table entry is empty, halt, indicating that the item is not in the table.
3. If the table entry matches the item, halt, indicating the index of the match.
4. If the table entry is neither empty nor equal to the item, then generate a new address for another probe of the table. (The algorithm for generating addresses is discussed later.)
5. Go to step 2.

The interesting aspect of this hash search algorithm is that during a search of the table items with many different hash codes may be encountered. The hash code merely determines where to start the search, and may possibly influence the sequence in which addresses are examined

(see Exercise 11.7). Because there is no pointer information, the search cannot be directed to a specific set of items that share a single hash code.

When the table to be searched is sparsely populated, the algorithm behaves like the hash search algorithm of Example 11.6. We expect to match the item or find an empty cell in a few probes of the table. When the table becomes full or nearly full, the algorithm behaves more like the sequential search of Example 11.4. Since few cells are empty, each new probe will be directed to a nonempty cell with high probability, and the search will rarely reach a termination point. Thus when we use the algorithm without pointers, the table should be partially empty—say, no more than 90 to 95 percent full. The unused memory in the table represents parts of the memory requirements of the algorithm. As such, it partially negates the memory savings achieved by eliminating pointers.

The algorithms for searching a table and for entering a new item into a table are given below.

EXAMPLE 11.7

By convention, if TABLE[i] contains a 0, then that element of the table is empty. The principal difference between this algorithm and Example 11.6 is the technique for generating the next address to search. In Example 11.6 we used the pointers in the data structure. Here we call the function NEXTADDRESS, which when given the address of the present search address, produces the next address to be examined.

```
procedure hashwithoutpointers(ITEM,PLACE);
integer ITEM, PLACE;
begin comment if TABLE[i] contains a 0, then no item has been placed there;
    integer i;
        PLACE := 0;
        i := HASH(ITEM);
    loop:
        if TABLE[i] = 0 then go to exit;
        if TABLE[i] ≠ ITEM then
            begin i := NEXTADDRESS(i);
                go to loop;
            end;
        PLACE := i;
        exit:
end procedure hashwithoutpointers;
```

```
procedure ENTER(ITEM);
integer ITEM;
begin integer i;
    i := HASH(ITEM);
    comment search for an empty place in the table and place ITEM in the
    first space available;
loop:
    if TABLE[i] ≠ 0 then
        begin comment look for a new place to insert ITEM;
            i := NEXTADDRESS(i);
            go to loop;
        end;
    TABLE[i] := ITEM;
end procedure ENTER;
```

The only part of the algorithm left unspecified is the procedure NEXTADDRESS. One candidate for this function that has often been used is

```
i := (i+1) MOD TABLESIZE;
```

Thus the array is searched sequentially, beginning at HASH(ITEM). The MOD operation causes the indexing to continue at 0 when i reaches the last element in the table. Other functions with better statistical properties are mentioned below.

To analyze this method of hash searching we shall make some assumptions that simplify the analysis, but which are not generally valid. Thus our analysis will not be rigorously correct, but it will be an accurate lower bound on the average computation.

Let α, the *load factor*, be the ratio of the number of entries in the table to the total size of the table. Then with probability α the first probe of the table will find a nonzero entry, and with probability $1 - \alpha$ the first probe will find an empty entry. Suppose that we wish to search for a given item. How many probes are necessary to determine that the item is not in the table? This is the number of probes required to find an empty cell. The algorithm terminates after i probes of the table when the first $i - 1$ probes reach nonempty cells and the ith probe reaches an empty cell.

If we assume that the probability of finding an empty cell is the same for any probe in a sequence of probes, then the probability of finding an empty cell on the ith probe is $\alpha^{i-1}(1 - \alpha)$. (This assumption is normally violated because the probability of finding an empty cell during any probe is not statistically independent of the outcomes of the previous

probes.) Our assumption leads to the following asymptotic formula for E, the average number of probes to search for an item not in the table:

$$\begin{aligned} E &= \sum_{i=1}^{\infty} i\alpha^{i-1}(1-\alpha) \\ &= (1-\alpha)\sum_{i=1}^{\infty} i\alpha^{i-1} \\ &= \frac{1-\alpha}{(1-\alpha)^2} = \frac{1}{1-\alpha} \end{aligned}$$

To determine the average number of probes to find an item in the table, we note that it takes exactly as many probes to retrieve an item as it took to insert that item into the table initially. Hence the average we desire is equal to the average number of probes required to fill the table to the load factor α. This average is

$$\frac{1}{\alpha}\int_0^{\alpha} \frac{1}{1-x}\,dx = -\frac{1}{\alpha}\log_e(1-\alpha) \cong 1 + \frac{\alpha}{2} + \frac{\alpha^2}{3} + \cdots$$

There are various ways that we can compare the behavior of Example 11.7 with Example 11.6. The most important point of comparison is related to the behavior when the table becomes full. As α approaches 1, the average number of probes required by Example 11.7 increases rapidly, approaching the size of the table. However, for Example 11.6, the average number of probes is $2/N$th the number of entries in the table. For smaller values of α, particularly for α less than $\frac{3}{4}$, the number of probes required by Example 11.7 is sufficiently small to make the algorithm competitive with Example 11.6.

The analysis of Example 11.7 relies heavily on the assumption that the outcomes of successive probes are statistically independent. The NEXTADDRESS function given above, which effectively causes the table to be searched linearly, will lead to a violation of our assumption, so that the algorithm will be less efficient than our analyses indicate. Clusters of items tend to accumulate in the table, so that if a probe hits somewhere in the cluster, a large number of probes is required to move the search beyond it.

Other functions for NEXTADDRESS have been reported in the literature. The functions that lead to the most efficient behavior are those which tend to randomize the address sequences produced by NEXTADDRESS. Maurer (1968) and Bell (1970) report methods based on

quadratic equations that have good randomization properties. One good method, due to Bell and Kaman (1970), is called the *linear-quotient method*. In this method the initial probe is made at the index given by the hash code, but successive probes are made at addresses that depend on the value of ITEM. Thus two items with the same hash code will generally produce different probe sequences, which tends to eliminate the clustering problem.

In the linear-quotient method we assume that the hash code is ITEM MOD TABLESIZE, and the probe sequence depends on ITEM ÷ TABLESIZE. That is, we use both the remainder and the quotient of the integer division of ITEM by TABLESIZE. In many computers, such as System/360-370, the quotient and remainder are both computed during the execution of a divide instruction.

We obtain the next address in a probe sequence from the present address by adding the quotient (that is, ITEM ÷ TABLESIZE) to the present address and taking the result modulo the size of the table. Thus the sequence of probe addresses is

HASH(ITEM)
[HASH(ITEM)+quotient] MOD TABLESIZE
[HASH(ITEM)+2·quotient] MOD TABLESIZE
[HASH(ITEM)+3·quotient] MOD TABLESIZE

The sequence of addresses will cover the entire table if, and only if, TABLESIZE is a prime number (see Exercise 11.8).

EXAMPLE 11.8

The linear-quotient method is given below:

```
integer quotient, remainder;
integer procedure HASH(ITEM);
value ITEM;
integer ITEM;
    begin comment this procedure produces the address of the first probe
    and initializes variables for successive probes;
        HASH := remainder := ITEM MOD TABLESIZE;
        quotient := ITEM ÷ TABLESIZE;
        if quotient = 0 then quotient := 1;
    end procedure HASH;
integer procedure NEXTADDRESS(address);
```

```
value address;
integer address;
begin integer temp;
    temp := (address + quotient) MOD TABLESIZE;
    if temp = remainder then
        begin comment we have searched the entire table. Notify search
        routine by setting address to -1;
            NEXTADDRESS := -1;
        end
    else NEXTADDRESS := temp;
end procedure NEXTADDRESS;
```

The interested reader is referred to Bell (1970) and Bell and Kaman (1970) for the combinatorial properties of Example 11.8. The statistical behavior of the example when it is used in a search algorithm is closely approximated by our analysis.

Before closing this section we must consider one final point concerning hash searching with and without pointers. When pointers are used, as in Example 11.6, it is possible to delete items from the table by the standard techniques for deleting items from lists. The memory that holds the deleted item can then be linked to a list of available memory for reuse. Search time diminishes when items are deleted. In Example 11.7 an item cannot be eliminated, leaving an empty cell in the table, because this empty cell can cause searches to stop prematurely. Thus items should be deleted by marking them with a special tag. New items can be entered in tagged cells, but searches cannot stop on these cells. Deleting items by tagging them does not improve search time and does not lead to efficient reuse of memory. Thus the hash search with pointers is more desirable than the search without pointers when items are to be deleted with high frequency from the table.

11.2.3 Height-balanced Trees

For many applications neither hash search nor binary search are suitable algorithms. In particular, there are many situations in which we wish to accomplish the following three processes efficiently:

1. Search for a specified item.
2. Enter a new item.
3. Delete a specified item.

The binary search is efficient for searching, but insertion and deletion of new items are extremely costly, since the table must be main-

tained in numerically ascending order. Hash searching with pointers is efficient for the last two items, but the efficiency of the search depends on the amount of memory that we wish to allocate. When tables become very large, the memory required for efficient searching can become excessive. Hash searching without pointers has a similar problem related to the inefficient use of memory. More important, the deletion process is a serious disadvantage, because the memory for deleted items cannot easily be returned to use, and deletions do not decrease search time.

A method for accomplishing all three of these processes efficiently has been developed by Adel'son-Vel'skii and Landis (1962). Foster (1965) provides a detailed analysis of their work and supplies the name *AVL tree* after the authors. Here we use the more descriptive term *height-balanced tree.*

To understand the height-balanced tree algorithm consider the binary tree shown in Figure 11.4*a*. In this tree we use the convention that the number stored at each node of the tree is smaller than those at the nodes in its right subtree and larger than those at the nodes in its left subtree. Thus we search for the number i as follows:

1. If the node contains the number i, halt.
2. If i is less than the number at the node, examine the left successor link, otherwise examine the right successor link.
3. If the link is empty, halt, indicating that the number i is not in the tree, otherwise follow this link to the successor node and return to step 1.

Figure 11.4*a* suggests that, on the average, we would expect to do approximately $\lfloor \log_2 N \rfloor$ comparisons if there are N items in the tree. Unfortunately, Figure 11.4*b* shows a case for which the worst case requires N comparisons when there are N items in the tree. Since there is a large difference between $\log_2 N$ and N, we wish to prevent trees from becoming badly unbalanced, as in Figure 11.4*b*, as we insert items into the tree. The height-balanced tree insertion algorithm does indeed accomplish this. The longest path in a height-balanced tree is guaranteed never to exceed $K \log_2 N$, where K is a small constant.

Specifically, we define a height-balanced tree as one that has the following property:

Height-balanced Tree For every node of a height-balanced tree, the length of the longest path in the left subtree differs from the length of the longest path in the right subtree by no more than one branch.

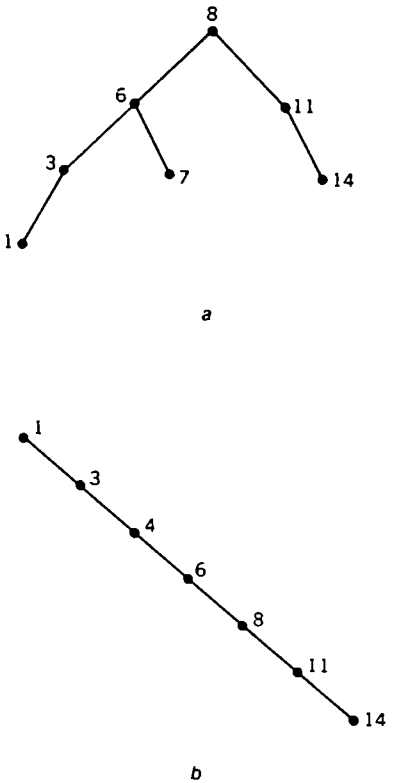


FIGURE 11.4 Two search trees: (*a*) a nearly balanced tree and (*b*) a badly balanced tree

The tree in Figure 11.4*a* is a height-balanced tree; the tree in Figure 11.4*b* is not.

First we shall see how to maintain trees that satisfy the balance property with successive insertions, and then we shall see how both the insertions and the searching can be done in a time proportional to $\log_2 N$. The deletion algorithm is the subject of Exercise 11.4.

Let us assume that a new item inserted into the tree is always inserted as a leaf. To do this we trace a path through the tree until we find the predecessor node of the item to be inserted. Typical cases are shown in Figure 11.5. The insertion may cause the tree to fail to satisfy the balance property. To help us detect such a failure, at each node in the tree we include information about the longest paths in the two subtrees of the node. A node is said to be *balanced* if the longest paths in each of its two subtrees are equal. If the longest path in the left subtree is one longer than the longest path in the right subtree, then we say that the node is *left heavy;* we define *right heavy* similarly. Every node is in one of these three states.

When an item is added to a height-balanced tree, one or more nodes

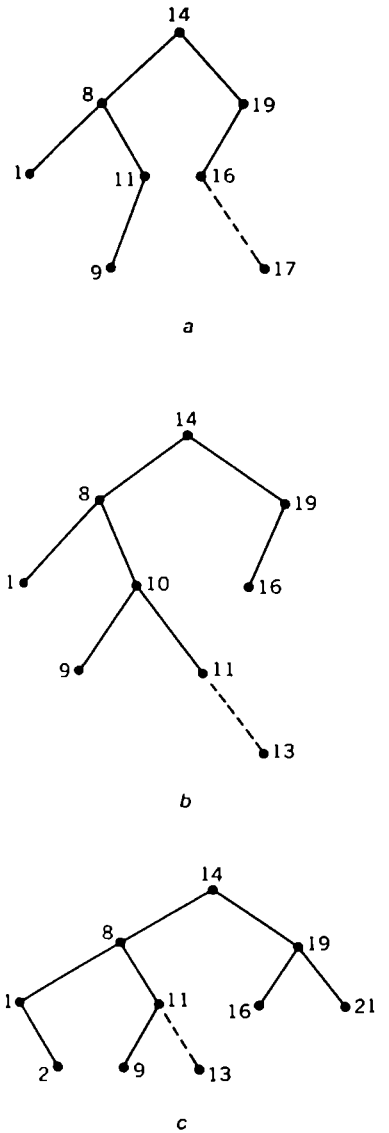


FIGURE 11.5 Insertion of new nodes into height-balanced trees. (*a*) Insertion of node 17 makes node 19 a critical node. (*b*) Insertion of node 13 makes node 8 a critical node. (*c*) Insertion of node 13 leaves no node critical

in the tree will change state. Examination of Figure 11.5 shows that all the nodes that do change state are on the path between the root of the tree and the new node of the tree. One of three changes of state may occur to a node on that path:

1. The node was formerly balanced and has become left heavy or right heavy.

2. The node was formerly heavy in one direction and has become balanced.

3. The node was formerly left heavy or right heavy, and the new node was inserted in the heavy subtree.

If condition 1 applies to a node, then the predecessor of this node undergoes a change of state. If condition 2 applies, then the predecessor node does not change state, nor does any other node above the predecessor node change state, because the length of the longest path in the subtree is unchanged. If condition 3 applies, then the node no longer satisfies the balance property. Such a node is said to be a *critical node.*

These observations suggest that a suitable insertion algorithm should check predecessors on the path to the root to see which of the three conditions applies. When condition 2 applies, the algorithm changes the state of the node and exits. When condition 1 applies, the algorithm changes the state of the node and then checks the predecessor node. When condition 3 applies, the algorithm must rearrange the tree. If the rearrangement of the tree returns the critical node to its original state, then no node in the tree above the critical node need be examined.

The next problem concerns tree rearrangement when a critical node is encountered. How might this be done efficiently to leave the tree in balanced form? Figure 11.6 illustrates the three different cases that might occur and the rearrangements that bring these trees into balanced form. In the first case the tree rooted at the critical node has only three nodes after the new item is inserted. The new item becomes the root of the tree after rearrangement. In the second case the node marked N is the new node and the boxes labeled S_1, S_2, and S_3 are each trees. The number in parentheses indicates the length of the longest path in these trees after node N is inserted. Since A is the critical node in this example, node B originally must be in the balanced state. The second case covers the situation when B becomes heavy in the same direction in which A is heavy. The third case is similar to the second, except that B becomes heavy in the direction opposite from that in which A is heavy. For this case it must be that node C is originally in the balanced state. The new node can be added to either subtree of C. Figure 11.6 shows the new node added to the left subtree, but the algorithm in Example 11.9 also treats the instance in which the new node is added to the right subtree. Note that the tree-rearranging computation involves the modification of no more than four pointers in the worst case. The complete details of the algorithm are given in Example 11.9.

To prove that the height-balanced tree algorithm is as efficient as claimed, we need only show that the longest path in the tree does not exceed $K \log_2 N$ for a tree with N nodes, where K is a small constant. If

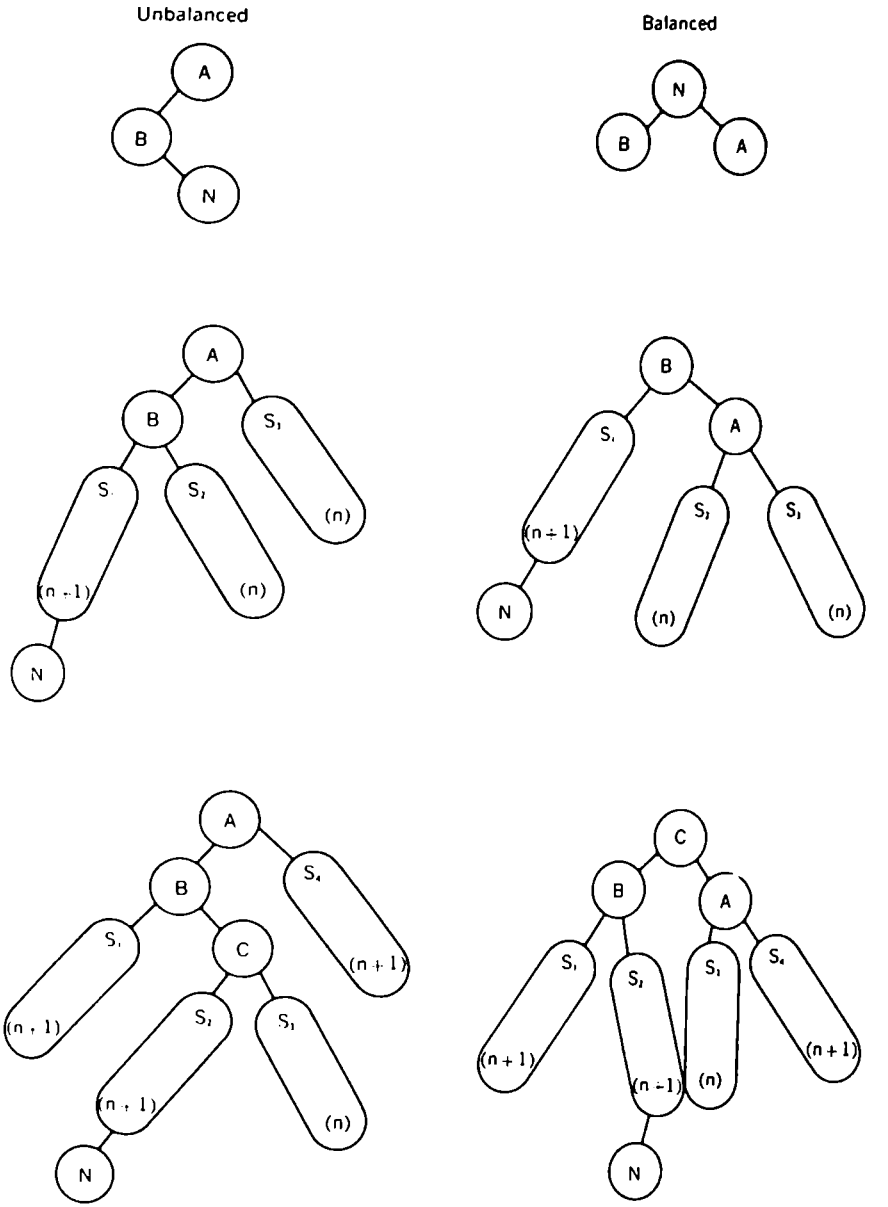


FIGURE 11.6 Modifications at a critical node that restore the height-balanced property

this is the case, then neither the search process nor the insertion process can exceed a number of steps proportional to $\log_2 N$. Although we do not include the deletion algorithm in Example 11.9, the hint given in Exercise 11.4 suggests that deletion also requires a number of steps proportional to $\log_2 N$.

EXAMPLE 11.9

```
integer balanced, left, right;
comment these integers have values 0, 1, and 2, respectively;
integer array nodevalue[1:n], nodeweight[1:n],
    link[1:n,1:2], path[1:m], direction[1:m]
comment     nodevalue    contains the value associated with a node.
            nodeweight   contains the state of the node. The value 0 in-
                         dicates balanced, 1 indicates left heavy, and
                         2 indicates right heavy.
            link         contains pointers to the left and right subtrees
                         of a node. The left subtree of node i is in
                         link[i,1].
            path         contains pointers to nodes that form a path
                         from the root to the point of insertion of a node.
            direction    shows the direction of the path. The node at
                         path[i] is always a left or right successor of
                         path[i—1]. The value of direction[i—1] is 1 or 2
                         to indicate path[i] is a left or right successor,
                         respectively;
integer root;
comment root is a pointer to the root node of the tree;
integer level;
integer procedure opposite(dir);
value dir;
integer dir;
    if dir = left then opposite := right else
    opposite := left;
procedure changehead(j);
value j;
integer j;
begin comment this procedure does a portion of the link rearranging re-
quired to maintain the balance property. It links the subtree rooted at the
critical node to the rest of the tree. Special consideration is given to the case
in which the critical node is the root of the entire tree;
    if level = 1 then
        begin comment the critical node is the root of the tree, and the new
        root is the node at position j on the path;
            root := path[j];
        end
    else
    begin comment in the normal case the node at path[level—1] is made to
    have the node at path[j] as its successor. The direction of the pointer to
    the successor is given in direction[level—1];
```

```
        link[path[level−1],direction[level−1]] := path[j];
    end;

end procedure changehead;

procedure exchange(a,b);

value a,b;

integer a,b;

begin comment this procedure changes the links of a pair of nodes in the
tree as part of the process of making the tree into a height-balanced tree;
    link[path[a],direction[a]] := link[path[b],opposite(direction[a])];
    link[path[b],opposite(direction[a])] := path[a];

end procedure exchange;

procedure savepath(place,dir);

value place, dir;

integer place, dir;

begin comment this procedure places entries in the arrays path and direc-
tion as a path is traced from the root to a point of insertion;
    level := level + 1;
    path[level] := place;
    direction[level] := dir;

end procedure savepath;

integer procedure search(item);

value item;

integer item;

begin comment this procedure searches for an item in the tree. If the item
is not in the tree, the procedure returns the value 0. If the item is in the tree,
the procedure returns the index of the node that holds the item;
        integer where;
        where := root;
        level := 0;
    testequal:
        if where = 0 then go to exit;
        comment the tree is empty or we have reached a leaf if the previous
        test is satisfied;
        savepath(where, if nodevalue[where] ≥ item then left else right);
        if nodevalue[where] = item then go to exit;
        comment the preceding test is satisfied if the item is found in the
        tree;
        where := link[where,direction[level]];
        go to testequal;
    exit:
        search := where;

end procedure search;

procedure insertnode(item);
```

```
value item;

integer item;

begin comment this procedure inserts a node into the tree. If the new tree
fails to satisfy the balance property, then the procedure rearranges the link-
ages at the critical node to make the tree into a height-balanced tree;
    integer where, weight;
    where := search(item);
    if where ≠ 0 then go to exit;
    comment the branch to exit is taken when the item is already in tree;
    where := avail;
    comment avail is a procedure that returns the index of an available cell
    for a new tree node. The following statements enter the item into the
    new tree node;
    nodevalue[where] := item;
    link[where,left] := link[where,right] := 0;
    nodeweight[where] := balanced;
    savepath(where,balanced);
    comment the following statement links the new node into the tree;
    changehead(level);
    comment in the following loop we check and modify the state of each
    node on the path to the root. If a critical node is found, the branch is
    taken to the statement labeled move;
    for level := level—1 step —1 until 1 do
        begin weight := nodeweight[path[level]];
            if weight = direction[level] then
                begin comment node is critical;
                    go to move;
                end;
            nodeweight[path[level]] := direction[level];
            if weight ≠ balanced then
                begin comment a node has been found that has changed
                into a balanced state. Record this fact and exit the loop and
                the procedure;
                    nodeweight[path[level]] := balanced;
                    go to exit;
                end;
        end for loop;
    go to exit;

move:
    if direction[level] = direction[level+1] then
        begin comment this is the second case;
            changehead(level+1);
            exchange(level,level+1);
            nodeweight[path[level]] := nodeweight[path[level+1]] :=
                balanced;
        end
    else
        begin comment the next four statements apply to both cases 1 and 3;
            changehead(level+2);
            exchange(level,level+2);
```

```
                exchange(level+1,level+2);
                nodeweight[path[level+2]] := balanced;
                if direction[level+1] = direction[level+2] then
                    begin comment one possibility for the third case;
                        nodeweight[path[level]] := balanced;
                        nodeweight[path[level+1]] := opposite(direction
                            [level+1]);
                    end
                else
                if direction[level+2] ≠ balanced then
                    begin comment the other possibility for the third case;
                        nodeweight[path[level]] := opposite(direction[level]);
                        nodeweight[path[level+1]] := balanced;
                    end
                else
                    begin comment the first case;
                        nodeweight[path[level]] := nodeweight[path[level+1]]
                            := balanced;
                    end;
            end;
        exit:
        end procedure insertnode;
```

For the proof we construct a sequence of trees $T_1, T_2, T_3, \ldots$ such that the longest path in tree T_i is length i and no height-balanced tree with a path of length i has fewer nodes than T_i. Figure 11.7 shows T_1, T_2, T_3, and T_4. To construct T_i we construct a tree with T_{i-2} and T_{i-1} as the left and right subtrees, respectively, of a single node. This is shown in Figure 11.7. For each i, T_i has the balance property. Moreover, if we delete a node from T_i, either the derived tree will no longer have the balance property, or the longest path in the derived tree will not be of length i. This can be shown by an inductive argument which is left as an exercise.

Each of the trees we construct is a height-balanced tree that is more nearly unbalanced than any height-balanced tree with an equal number of nodes. Thus each tree in the sequence has the longest path of any height-balanced tree with an equal number of nodes. We shall count the number of nodes in each of these trees and establish the relation between the length of the longest path and the number of nodes in the tree.

Let $t(i)$ be the number of nodes in T_i. Then Figure 11.7 indicates that the following relations hold:

$$t(1) = 2$$

$$t(2) = 4$$

$$t(i) = t(i-1) + t(i-2) + 1 \qquad \text{for } i > 2$$

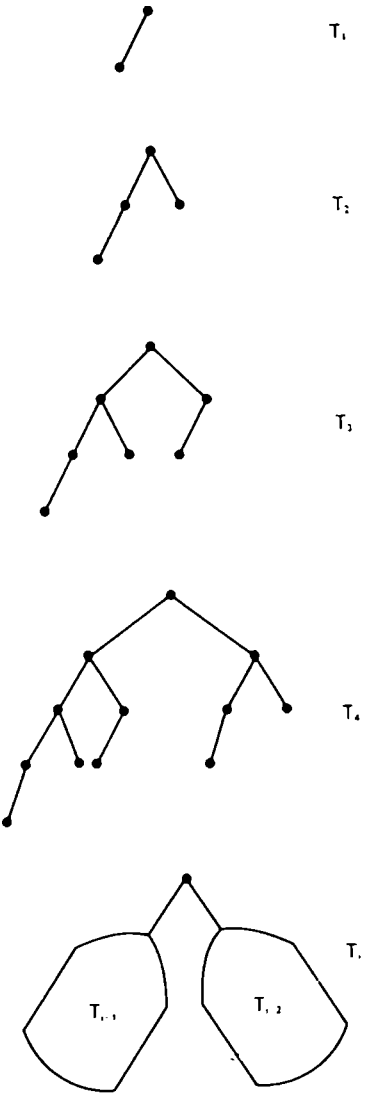


FIGURE 11.7 The sequence of trees T_1, T_2, T_3, . . .

These relations are very similar to the relations that define the Fibonacci numbers (cf. Knuth 1968):

$$F(0) = 0$$

$$F(1) = 1$$

$$F(i) = F(i-1) + F(i-2) \qquad \text{for } i > 1$$

Close inspection of the first few terms in the series reveals that

$$t(i) = F(i + 3) - 1$$

This last equation can be shown to be an identity by substituting for $t(i)$ in its defining equation. It is well known that the Fibonacci numbers satisfy the asymptotic formula (cf. Knuth 1968)

$$F(i) \cong \frac{\phi^i}{\sqrt{5}}$$

where

$$\phi = \frac{1 + \sqrt{5}}{2}$$

From this we obtain

$$t(i) \cong \frac{\phi^{i+3}}{\sqrt{5}} - 1$$

Now we solve for i in terms of $t(i)$. For this we find

$$\phi^{i+3} \cong \sqrt{5}\,[t(i) + 1]$$
$$i + 3 = \log_\phi \sqrt{5} + \log_\phi [t(i) + 1]$$

From the identities

$$\log_\phi x = \log_2 x / \log_2 \phi$$

and

$$\log_2 \phi = .694$$

we obtain the bound

$$i < 3/2 \log_2 [t(i) + 1] - 1$$

Since $t(i)$ is the number of nodes in the tree, we see that for a tree with N nodes, the longest path cannot exceed $\frac{3}{2} \log_2 (N + 1)$, which is the result we set out to prove.

Foster (1973) has proposed an interesting generalization of height-balanced trees in which the height-balance property is relaxed from that given here. More specifically, we say a tree is an $HB[k]$ tree for a given integer k if, for every node in the tree, the height of the left subtree differs at most by k from the height of the right subtree. What we have called height-balanced trees in the previous discussion are $HB[1]$ trees by this definition. The motivation of this generalization is that we permit a slightly greater unbalance to exist in the tree, with the hope that fewer rebalancings are necessary as the tree is altered. If k is a small integer, the length of the longest path in the tree remains proportional to $\log_2 N$, so that the average time to search an $HB[k]$ tree is not too much greater than the time to search an $HB[1]$ tree. Foster's experimental data suggest that, as k increases from 1 to 4, the average length of a search of an $HB[k]$ tree increases by one probe but the average number of tree rearrangements decreases by more than a factor of 6. Thus $HB[k]$ trees appear to be a good candidate data structure for large data bases.

REFERENCES

Other sorting algorithms exist that have the same computational behavior as Floyd's tree sort. In particular, Hoare's quicksort (1961) and its later modifications (Singleton 1969) are algorithms for sorting in a time proportional to $N \log_2 N$ that can be more efficient than Floyd's tree sort in many contexts.

Hash addressing, like so many other algorithms in computer lore, appears to have been invented early in the history of computers and is not traceable to a particular individual. Dumey (1956) describes the use of hash addressing for maintaining data files. The first detailed analysis of hash addressing appears in an article by Peterson (1957). The definitive book on searching and sorting is by Knuth (1973).

EXERCISES

11.1 Give an algorithm for constructing a permutation of the integers 1 to N when given a ranking sequence for that permutation.

11.2 Simulate the tree-sort algorithm on an array containing 15 integers 1, 2, . . . , 15. Assume that the data are initially sorted in ascending sequence so that A[i] contains i.

11.3 Assume that the hash codes for data are integers uniformly distributed in the interval $0 \leq$ hash code $\leq N - 1$. If an array to be searched contains M items, show that the average length of a list of items with the same hash code contains M/N items. (Hint: Assume that the probability that an item has hash code h_0 is $1/N$, and the probability is $1 - 1/N$ that the item has some other hash code. Show that the length of the list of items with hash code h_0 is binomially distributed.)

11.4 Write an algorithm that deletes a specified item from a height-balanced tree. (Hint: If the item is not located in a leaf, find its postorder predecessor and see if it can be deleted.)

11.5 Prove the following assertion:

If a leaf node is removed from T_i, then either the new tree fails to satisfy the balance property or the longest path in the new tree has a length less than i.

11.6 Binary searching is not efficient when new items have to be added to the table. A programmer suggests that the table be stored as a linked list rather than as a contiguous array in order to facilitate insertion of new items. In this case how many steps are required in the worst case to search for an item? How many steps are required to enter a new item, given the position at which it is inserted?

11.7 Let TABLESIZE = 13 in Example 11.8. Show the sequence of addresses probed for quotients with the values 1, 3, and 4. For each sequence assume a remainder of 0.

11.8 In Example 11.8 prove that the entire table is searched if, and only if, the value of TABLESIZE is prime. [Hint: If probes i and j reach the same address, $j > i$, then $(j - i) \cdot$ quotient is a multiple of TABLESIZE.]

APPENDIX

EXPLANATION OF THE ALGOL-LIKE NOTATION

The notation in this book is an extension of ALGOL 60 that includes coroutines. This appendix contains a short summary of the notation. A full report on the revised ALGOL 60 language appears in *Comm. ACM*, vol. 6, no. 1, January 1963, pp. 1–23.

A.1 BASIC SYMBOLS

The basic entities from which statements are constructed are

Constants
Identifiers
Reserved words

Constants have real, integer, or boolean values, and these values cannot change during program execution. Examples are:

Real constants	1.2 3.14159268 −7.99983
Integer constants	1, 2, 3, . . . , −1, −2, . . .
Boolean constants	**true, false**

Identifiers are user-defined names. The meaning of an identifier depends on the context in which it is used. The various meanings that can be assigned to identifiers are described in later sections. Identifiers are strings of alphanumeric symbols that begin with a letter. Examples of identifiers are:

```
A   I   J   MAX   quotient   notfound
```

Reserved words are words that have a fixed meaning in programs. They are printed in boldface. The meaning of each reserved word is discussed in later sections. Examples of reserved words are:

```
begin   end   procedure   if   then   else
```

A.2 SIMPLE VARIABLES

Simple variables are named entities that occupy one unit of memory storage. They can be assigned values during the course of computation, and any particular simple variable may have its value changed by assigning it a new value. Declaration statements for simple variables are used in programs to reserve space for the variables and to indicate their type. Variables declared to be boolean can assume only the values **true** and **false**. Similarly, integer variables can attain only integer values, and real variables can attain only real values. Examples of declarations are:

```
real      X, Y, Z;
integer   I, J, MAX;
boolean   found, erroroccurred;
```

A.3 ARRAYS

Arrays are data structures in which individual elements are specified by subscripts. A one-dimensional array X with lower bound LB and an upper bound UB is a data structure that consists of the variables X[LB], X[LB+1], . . . , X[UB]. That is, X is a vector in which the least index is LB and the greatest index is UB. Higher-dimensional arrays are data structures which have two or more indices. The two-dimensional matrix M, with rows indexed from LB1 to UB1, and columns indexed from LB2 to UB2, consists of a set of variables M[I,J] for all pairs I,J such that LB1 $\leq$ I $\leq$ UB1 and LB2 $\leq$ J $\leq$ UB2.

The size, dimension, and data type of an array are specified in array declarations. Examples of array declarations are:

```
real array      A[LB1 : UB1, LB2 : UB2];
integer array   X[LB : UB],  Y[1 : 10];
boolean array   V[0 : 7, 0 : 7, 0 : 7];
```

A.4 EXPRESSIONS

Values of variables, array elements, and constants can be combined by arithmetic and boolean operators to produce new values. Expressions fall into two categories:

1. Arithmetic expressions (combinations of entities of type real and integer)
2. Boolean expressions (combinations of boolean variables, boolean constants, and arithmetic relations)

Arithmetic expressions are constructed from the following operators:

ARITHMETIC OPERATOR	MEANING
+	Addition
−	Subtraction
*	Multiplication
/	Division
÷	Integer division
MOD	Modulus, the remainder after an integer division
↑	Exponentiation

Examples of arithmetic expressions are:

A+B*(C/D)↑2 (N MOD TABLESIZE)+1

The arithmetic expressions can be compared to each other in magnitude by using a construction known as an arithmetic relation. The relation involves an operator which can be any of the following:

$=$ $\neq$ $>$ $<$ $\geq$ $\leq$

When an arithmetic relation is evaluated during program execution it has a value of either **true** or **false**, depending on whether or not the relation holds. Examples of arithmetic relations are:

B = A+6 A*4 < B−1

When A and B have the values 1 and 7, respectively, both relations hold, and are said to have the value **true**. When A and B have the value 5 and 6, respectively, both relations have the value **false**.

Boolean expressions are combinations of relations, boolean variables, and boolean constants with the following boolean operators:

NOT AND OR EXCLUSIVE OR

The operators are defined by the table below:

X	Y	NOT X	X AND Y	X OR Y	X EXCLUSIVE OR Y
false	**false**	**true**	**false**	**false**	**false**
false	**true**	**true**	**false**	**true**	**true**
true	**false**	**false**	**false**	**true**	**true**
true	**true**	**false**	**true**	**true**	**false**

Examples of boolean expressions are:

((NOT X) AND (I < J)) OR (I = 3*K) Z AND ((I ≠ J) OR (I > K + 2))

In these examples we assume that I, J, and K are arithmetic variables and X and Z are boolean variables.

A.5 ASSIGNMENT STATEMENTS

The values of variables can be changed by means of assignment statements. We use the symbol ← (and :=, its ALGOL version) to indicate

the transfer of data. The variable to the left of the arrow represents the name, or the memory location, of the variable. The expression to the right of the arrow is evaluated, and its value is placed in the indicated memory location. Examples of assignment statements are:

```
A[I] ← 3+X/Y[2,6];
B := (5*N) MOD J;
FOUND := false;
FOUND := NOT FOUND AND ITEMSEEN;
```

A.6 CONTROL STATEMENTS

The three types of control statements are the unconditional branch, the conditional statement, and the iteration.

An unconditional branch is a statement of the form

```
go to targetlabel;
```

where targetlabel is the label of some statement in the program. Examples of unconditional statements are:

```
go to compute;
. . .
compute: x:=y;
go to skiponestatement;
next: a := b;
skiponestatement: c := d;
```

A conditional statement can either execute or skip a statement, depending on the value of a boolean expression. Conditional statements have one of the following two forms:

```
if booleanexpression then S else T;
if booleanexpression then S;
```

Here S and T are statements. In both forms, if the boolean expression has the value **true**, then statement S is executed, otherwise statement S is skipped. In the first form of the conditional statement statement T is executed if, and only if, the boolean expression has the value **false**.

Examples of conditional statements are:

```
if I ≠ J then X[I] := X[I]+1;
If I > 3 then I := I − 1 else go to finish;
if pagenum = 0 then M := 0 else M := P;
```

Iteration statements are statements of the general form

```
for LOOPVARIABLE := FIRSTVALUE step INCREMENT
    until LOOPLIMIT do S;
```

Here S is a statement. The iteration statement causes S to be executed zero or more times, depending on the values of FIRSTVALUE, LOOPLIMIT, and INCREMENT. If INCREMENT is positive, then S is executed iteratively, with LOOPVARIABLE assuming the sequence of values FIRSTVALUE, FIRSTVALUE + INCREMENT, FIRSTVALUE + 2·INCREMENT . . . , FIRSTVALUE + K·INCREMENT, where K is the largest integer for which FIRSTVALUE + K·INCREMENT does not exceed LOOPLIMIT. In this case if FIRSTVALUE is greater than LOOPLIMIT, then S is not executed at all. The value of INCREMENT can also be negative, in which case the values assumed by LOOPVARIABLE decrease instead of increase. The iteration terminates when the value of LOOPVARIABLE falls below LOOPLIMIT. FIRSTVALUE, INCREMENT, and LOOPLIMIT need not be simple variables or constants, but can be arbitrary arithmetic expressions. The expressions are reevaluated before each execution of S, so that the execution of S can change these values and affect the iteration statement.

Examples of iteration statements are:

```
for i := 1 step 1 until MAX do A[i] := 0;
for j := i+1 step m until MAX−m do X[j] := X[j+m];
```

A.7 COMPOUND STATEMENTS

Iteration and conditional statements are used to control the execution of individual statements. To control the execution of a group of two or more statements the statements can be collected together into a compound statement which can be treated as an individual statement. The general form of a compound statement is

```
begin S1;S2;S3; . . . ;Sn end
```

where $S_1, \ldots, S_n$ are each assignment, control, or compound statements.

Examples of compound statements are:

```
if I = J then
    begin if I ≠ K then
        begin I := 2;
              J := 1;
        end
```

```
    else
        K := 3;
    end;
for I := 1 step 1 until 10 do
    begin A[I] := 0;
          B[I] := B[I+1] + B[I];
    end;
```

In the first example the compound statement

```
begin I := 2;
      J := 1;
end
```

is executed when I = J and I ≠ K. In the second example the compound statement is

```
begin A[I] := 0;
      B[I] := B[I+1] + B[I];
end
```

This statement is executed 10 times as I assumes the values 1, 2, . . . , 10.

A.8 PROCEDURES

Procedures are the ALGOL version of subroutines. These are subprograms that are invoked from arbitrary points in a program and return control to the calling point. The entry point is always the first statement of the procedure.

The least complex form of procedure is the procedure without parameters. It has the general form

```
procedure proc; S;
```

where S is a statement. Wherever the procedure call statement proc; appears in a program the effect of the call is the same as replacing the call by the statement S.

An example of a parameterless procedure is

```
procedure BUMPI;
    begin I := I+1;
        if I > MAX then go to error;
    end procedure BUMPI;
```

A typical program that uses this procedure might have the following appearance:

```
        ...
        if x = y then BUMPI;
        ...
        BUMPI;
        ...
    error:
        ...
```

The use of parameters for procedures greatly increases their utility. Procedure parameters are specified as part of the procedure declaration. Dummy parameters called *formal parameters* are used for the declaration, and these parameters are replaced by *actual parameters* when the procedure is invoked.

An example of a procedure with parameters is:

```
procedure BUMP (index, maxval, errorlabel);
integer index, maxval;
label errorlabel;
begin index := index+1;
    if index > maxval then go to errorlabel;
end procedure BUMP;
```

The formal parameters of this procedure are index, maxval, and errorlabel. The statements

```
integer index,maxval;
label errorlabel;
```

are specification statements that describe the type of the formal parameter. Specification statements differ from declarations in that declarations cause space to be reserved for the entity declared and specifications describe entities for which space has already been reserved. Typical calls on BUMP are:

```
        integer I,J,M,N;
        ...
        BUMP(I,N,error);
        ...
    error:
        ...
        BUMP(J,M,stop);
        ...
    stop:
        ...
```

Array specifications do not include the lower and upper bounds of the array. An example of array specification is

```
procedure SEARCH(ITEM,A);
integer ITEM;
integer array A;
   . . .
```

Parameters can be transmitted either by name or by value. When the value is transmitted, the actual parameter is evaluated at the time of the call, and the resulting value is the parameter manipulated by the procedure. When the name of the parameter is transmitted, the resulting computation is what would be obtained if all occurrences of the corresponding formal parameter name were replaced by the actual parameter (see Chapter 7 for a complete discussion of this point). Value parameters are indicated in the specification part. Parameters that are not specified as value parameters are name parameters by default.

An example of value and name parameters is

```
procedure change (N,I);
value N;
integer N,I;
   . . .
```

In this procedure N is called by value and I is called by name.

Procedures can transmit values back to the calling program through parameters called by name. Another convenient method of returning values is to treat the procedure as a function. Examples of this method are the procedures SINE(X) and SQRT(X) for computing the sine and square root of a number. Function procedures are specified by indicating a type for the procedure in its declaration. The functional value is transmitted back to the calling program by assigning a value to the procedure name in the body of the procedure.

An example of a function procedure is:

```
integer procedure BUMP(I);
integer I;
begin
BUMP := I+1;
end procedure BUMP;
```

A typical call on this procedure is

```
J := BUMP(I);
```

Function procedures need not have parameters. The rules for transmitting parameters to function procedures are identical to the rules for transmitting parameters to ordinary procedures.

A.9 COROUTINES

Coroutines are subprograms that are entered at the point of their last exit rather than at a beginning point. The initial entry to a coroutine is made to the first statement in the coroutine. An example of coroutine declarations is

```
coroutine X;
    begin
        . . .
        RESUME Y;
        . . .
        RESUME Y;
        . . .
        RESUME Y;
end coroutine X;
coroutine Y;
begin
    . . .
    RESUME X;
    . . .
    RESUME X;
    . . .
end coroutine Y;
```

We use RESUME to indicate a coroutine call. In coroutine X the statement

```
RESUME Y;
```

causes control to pass from X to Y, entering Y at its last call to X. (If Y is entered for the first time, control passes to the beginning of Y.) When the statement

```
RESUME X;
```

is reached, control passes to the statement immediately following the last coroutine call to Y.

BIBLIOGRAPHY

ADEL'SON-VEL'SKII, G. M., AND E. M. LANDIS, 1962: An algorithm for the organization of information, *Dokl. Akad. Nauk SSSR, Mathemat.*, vol. 146, no. 2, 1962, pp. 263–266.

BARTON, R. S., 1961: A new approach to the functional design of a digital computer, *Proc. WJCC*, 1961, pp. 393–396.

BELL, C. G., AND A. NEWELL, 1971: "Computer Structures: Readings and Examples," McGraw-Hill Book Company, New York, 1971.

BELL, J. R., 1970: The quadratic quotient method: A hash code eliminating secondary clustering, *Comm. ACM*, vol. 13, no. 2, February 1970, pp. 107–109.

——— AND C. H. KAMAN, 1970: The linear quotient hash code, *Comm. ACM*, vol. 13, no. 11, November 1970, pp. 675–677.

BURKS, A. W., H. H. GOLDSTINE, AND J. VON NEUMANN, 1946: Planning and coding of problems for an electronic computing instrument, parts 1 and 2. Institute for Advanced Study, Princeton, N.J. Reprinted in Bell and Newell (1971).

CONWAY, M., 1963: Design of a separable transition compiler, *Comm. ACM*, vol. 6, no. 7, July 1963, pp. 396–408.

DAVIS, MARTIN, 1958: "Computability and Unsolvability," McGraw-Hill Book Company, New York, 1958.

DIJKSTRA, E., 1958: "Communication with an Automatic Computer," dissertation, Technische Hochschule Eindhoven, 1958.

———, 1968: Cooperating sequential processes, in F. Genuys (ed.), "Programming Languages," Academic Press, New York, 1968, pp. 43–112.

DUMEY, A. I., 1956: Indexing for rapid random access memory systems, *Computers and Automation*, vol. 5, no. 12, December 1956, pp. 6–9.

FLOYD, R. W., 1964: Algorithm 245, treesort 3, *Comm. ACM*, vol. 7, no. 12, December 1964, p. 701.

FOSTER, C. C., 1964: Information storage and retrieval using AVL trees, *Proc. ACM Natl. Conf.*, 1965, pp. 192–205.

———, 1973: A generalization of AVL trees, *Comm. ACM*, vol. 16, no. 8, August 1973, pp. 513–517.

HALEY, A. C. D., 1962: The KDF9 computer system, *Proc. AFIPS Conf. FJCC*, vol. 22, 1962, Spartan Books, Washington, D.C., pp. 108–120.

HOARE, C. A. R., 1961: Algorithm 63, partition, and algorithm 64, quicksort, *Comm. ACM*, vol. 4, no. 7, July 1961, p. 321.

INGERMAN, P. Z., 1961: Thunks, *Comm. ACM*, vol. 4, no. 1, January 1961, pp. 55–58.

KENT, W., 1969: Assembler language macro-programming, *Computing Surveys*, vol. 1, no. 4, December 1969, pp. 183–196.

KNUTH, D. E., 1968: "The Art of Computer Programming," vol. 1, "Fundamental Algorithms," Addison-Wesley, Reading, Mass., 1968.

———, 1973: "The Art of Computer Programming," vol. 3, "Searching and Sorting," Addison-Wesley, Reading, Mass., 1973.

LAMPSON, B., 1968: A scheduling philosophy for multiprocessing systems, *Comm. ACM*, vol. 11, no. 5, May 1968, pp. 347–360.

LEINER, A. L., 1952: Buffering between input/output and the computer, *Proc. EJCC*, 1952, pp. 22–30.

LIN, S., AND T. RADO, 1965: Computer studies of Turing machine problems, *J. ACM*, vol. 12, no. 2, April 1965, pp. 196–212.

ŁUKASIEWICZ, J., 1951: "Aristotle's Syllogistic from the Standpoint of Modern Formal Logic," Oxford University Press, Oxford, 1951.

MAURER, W. D., 1968: An improved hash code for scatter storage, *Comm. ACM*, vol. 11, no. 1, January 1968, pp. 35–38.

MINSKY, M., 1967: "Computation: Finite and Infinite Machines," Prentice-Hall, Inc., Englewood Cliffs, N.J., 1967.

MOCK, O., AND C. J. SWIFT, 1959: The SHARE 709 System, programmed input/output buffering, *J. ACM*, vol. 6, no. 2, April 1959, pp. 145–151.

MORRIS, ROBERT, 1968: Scatter storage techniques, *Comm. ACM*, vol. 11, no. 1, January 1968, pp. 38–44.

NEWELL, A. (ed.), 1961: "Information Processing Language V: Manual," Prentice-Hall, Englewood Cliffs, N.J., 1961.

——— AND J. C. SHAW, 1957: Programming the logic theory machine, *Proc. WJCC*, 1957, pp. 230–240.

PERLIS, A. J., AND C. THORNTON, 1960: Symbol manipulation by threaded lists, *Comm. ACM*, vol. 3, no. 4, April 1960, pp. 195–204.

PETERSON, W. W., 1957: Addressing for random access storage, *IBM J. Res. Dev.*, vol. 1, no. 2, April 1957, pp. 130–146.

ROGERS, H., JR., 1967: "Theory of Recursive Functions and Effective Computability," McGraw-Hill, New York, 1967.

ROSEN, S., 1969: Electronic computers: A historical survey, *Computing Surveys*, vol. 1, no. 1, March 1969, pp. 7–36.

ROSS, D. T., 1961: A generalized technique for symbol manipulation and numerical calculation, *Comm. ACM*, vol. 4, no. 3, March 1961, pp. 147–150.

SHAW, J. C., A. NEWELL, H. A. SIMON, AND T. O. ELLIS, 1958: A command structure for complex information processing, *Proc. WJCC*, 1958, pp. 119–128.

SINGLETON, R. C., 1969: Algorithm 347, *Comm. ACM*, vol. 12, no. 3, March 1969, pp. 185–186.

TARSKI, A., 1956: "Logic, Semantics, Metamathematics," Clarendon, Oxford, 1956.

TURING, A. M., 1936: On computable numbers with an application to the Entsheidungsproblem, *Proc. London Math. Soc.*, ser. 2, vol. 42, pp. 230–265.

VAN WIJNGAARDEN, A. (ED.), B. J. MAILLOUX, J. E. L. PECK, AND C. H. A. KOSTER, 1968: "Draft Report on the Algorithmic Language ALGOL 68," Mathematish Centrum, Amsterdam, 1968.

WILLIAMS, J. W. J., 1964: Algorithm 232, Heapsort, *Comm. ACM*, vol. 7, no. 6, June 1964, pp. 347–438.

WIRTH, N., 1969: On multiprogramming, machine coding, and computer organization, *Comm. ACM*, vol. 12, no. 9, September 1969, pp. 489–498.

——— AND C. A. R. HOARE, 1966: A contribution to the development of ALGOL, *Comm. ACM*, vol. 10, no. 6, June 1966, pp. 413–432.

INDEXES

INDEX

To IBM System/360-370 Instructions

CODE	DESCRIPTION	PAGE
A	Add full word	61
AD	Add floating-point double word	65
ADR	Add floating-point double word (register)	65
AE	Add floating-point full word	65
AER	Add floating-point full word (register)	65
AH	Add halfword	64
AR	Add (register)	63
B	Branch unconditionally	68
BAL	Branch and link	68
BALR	Branch and link (register)	68
BC	Branch on condition	67
BCR	Branch on condition (register)	67
BCT	Branch on count	68
BCTR	Branch on count (register)	68
BE	Branch on equal	68
BH	Branch on high	68
BL	Branch on low	68
BM	Branch on minus	68
BNE	Branch on not equal	68
BNH	Branch on not high	68
BNL	Branch on not low	68
BNM	Branch on not minus	68
BNP	Branch on not plus	68
BNZ	Branch on not zero	68

CODE	DESCRIPTION	PAGE
BO	Branch on overflow	68
BP	Branch on plus	68
BR	Branch unconditionally (register)	68
BZ	Branch on zero	68
C	Compare	65
CR	Compare (register)	65
D	Divide full word	61
DC	Define constant (pseudo-instruction)	124
DD	Divide floating-point double word	65
DDR	Divide floating-point double word (register)	65
DE	Divide floating-point full word	65
DER	Divide floating-point full word (register)	65
DR	Divide (register)	63
DROP	Drop base register (pseudo-instruction)	124
DS	Define storage (pseudo-instruction)	124
END	End of program (pseudo-instruction)	124
ENTRY	Entry symbol (pseudo-instruction)	136
EQU	Equate (pseudo-instruction)	124
EXTRN	External symbol (pseudo-instruction)	136
L	Load full word	61
LCR	Load (two's) complement (register)	63
LD	Load floating-point double word	64
LDR	Load floating-point double word (register)	64
LE	Load floating-point full word	64
LER	Load floating-point full (register)	64
LH	Load halfword	64
LR	Load (register)	63
LTR	Load and test (register)	65
M	Multiply full word	61
MACRO	Begin macro (pseudo-instruction)	141
MD	Multiply floating-point double word	65
MDR	Multiply floating-point double word (register)	65
ME	Multiply floating-point full word	65
MEND	End macro (pseudo-instruction)	141
MER	Multiply floating-point full word (register)	65
MR	Multiply (register)	63
N	Logical AND	96
NR	Logical AND (register)	96
O	Logical OR	97
OR	Logical OR (register)	97
ORG	Origin (pseudo-instruction)	124
READ	Read (channel command)	230

CODE	DESCRIPTION	PAGE
S	Subtract full word	61
SD	Subtract floating-point double word	65
SDR	Subtract floating-point double word (register)	65
SE	Subtract floating-point full word	65
SER	Subtract floating-point full word (register)	65
SH	Subtract halfword	64
SIO	Start input/output	231
SLA	Shift left arithmetic	102
SLDA	Shift left (double) arithmetic	102
SLDL	Shift left (double) logical	102
SLL	Shift left logical	102
SR	Subtract (register)	63
SRA	Shift right arithmetic	102
SRDA	Shift right (double) arithmetic	102
SRDL	Shift right (double) logical	102
SRL	Shift right logical	102
ST	Store full word	61
STD	Store floating-point double word	65
STE	Store floating-point full word	65
STH	Store halfword	64
STM	Store multiple	233
TIC	Transfer in channel (channel command)	230
TIO	Test input/output	231
USING	Using base register (pseudo-instruction)	124
WRITE	Write (channel command)	230
X	Logical EXCLUSIVE OR	97
XR	Logical EXCLUSIVE OR (register)	97

INDEX

To PDP-11 Instructions

CODE	DESCRIPTION	PAGE
ADC	Add carry	106
ADD	Add	24
.ASCII	Character convert (pseudo-instruction)	120
ASL	Arithmetic shift left	99
ASR	Arithmetic shift right	99
B	Suffix for byte instructions	166
BCC	Branch on carry clear	34
BCS	Branch on carry set	34
BEQ	Branch on equal (zero)	28
BGE	Branch on greater or equal (zero)	36
BGT	Branch on greater than (zero)	36
BHI	Branch higher	36
BHIS	Branch higher or same	49
BIC	Bit clear	95
BICB	Bit clear byte	168
BIS	Bit set (OR)	95
BISB	Bit set (OR) byte	168
BIT	Bit test (AND)	95
BLE	Branch on less than or equal (zero)	36
BLO	Branch lower	49
BLOS	Branch on lower or same	36
BLT	Branch on less than (zero)	36
BMI	Branch on minus	28
BNE	Branch on not equal (zero)	28

CODE	DESCRIPTION	PAGE
BPL	Branch on plus	28
BR	Branch (unconditional)	28
BVC	Branch on overflow clear	34
BVS	Branch on overflow set	34
.BYTE	Byte constants (pseudo-instruction)	120
CLC	Clear carry condition code	100
CLN	Clear negative condition code	100
CLR	Clear	39
CLV	Clear overflow condition code	39
CLZ	Clear zero condition code	100
CMP	Compare	35
COM	Complement	37
DEC	Decrement	28
.END	End of program (pseudo-instruction)	102
.ENDM	End of macro (pseudo-instruction)	140
.EVEN	Even (pseudo-instruction)	120
.GLOBAL	Global (pseudo-instruction)	136
HALT	Halt	79
INC	Increment	28
JMP	Jump	48
JSR	Jump subroutine	174
.MACRO	Begin macro (pseudo-instruction)	140
MOV	Move	24
MOVB	Move byte	166
NEG	Negate	37
ROL	Rotate left (with carry)	99
ROR	Rotate right (with carry)	99
RTI	Return from interrupt	215
RTS	Return from subroutine	175
SEC	Set carry condition code	100
SEN	Set negative condition code	100
SEV	Set overflow condition code	100
SEZ	Set zero condition code	100
SUB	Subtract	24
TST	Test	208
TSTB	Test byte	208
.WORD	Word constants (pseudo-instruction)	120
%	Assembler designation for register	117

CODE	DESCRIPTION	PAGE
. = . +	Block storage (pseudo-instruction)	120
.	Decimal (pseudo-instruction)	120
@	Prefix for deferred or indirect addressing	76
=	Direct assignment (equate) (pseudo-instruction)	120
. =	Origin (pseudo-instruction)	120

SUBJECT INDEX

Absolute address mode, 84
AC, 23, 25
Actual parameter, 187, 348
Address, 18–21
  byte, 18
  word, 20
Address modification, 74–94
  absolute, 84
  autodecrement, 81–83, 85
  autoincrement, 81–83, 85, 151–153
  deferred (*see* indirect *below*)
  displacement in page, 91–92
  immediate, 84
  index register, 80, 83–89
  indexing, 148–151, 156–158
  indirect, 76–79, 83, 85–86, 156–158
  page number, 91–92
  page registers, 91–93
  paging, 90–94
  PDP-11, 74–86, 92–93
  relative, 74–76, 83
  segmentation, 93
  System/360–370, 86–90, 93
Address polynomial, 155–156
Address word, 47, 75
Adel'son-Vel'skii, G. M., 327
ALGOL, 21, 153, 158, 165, 201, 264
  call by name (replacement rule), 195, 197–200
  call by value, 196
ALGOL-like language, 341–350
Algorithms, 4–8
  defined, 4, 8
  Turing machine, 8–13
AND, 94–97, 344
Aristotle, 6–7
Arithmetic instructions, 60–65
Arithmetic shift instructions, 98–104
Array, 148
  in ALGOL, 343
  bounds, 153–155
  multidimensional, 155–158
  one-dimensional, 148–155
  (*See also* Link list; Push-down stack; Queue)
Assembler, 113–142
  PDP-11, 114–123, 129–132, 139–140
  System/360–370, 124–128, 132, 141–142
  two-pass processing, 128–132
Assembly language (*see* Assembler)
Autodecrement, 81–82, 83, 85
Autoincrement, 81–82, 83, 85, 151–153
AVL tree (*see* Height-balanced tree)

B field, 69–70, 86–89
Balanced node, 328
Barton, Robert S., 295

Base address (of array), 155
Base field, 69–70, 86–89
Base register, 87–90
Bell, C. Gordon, 52, 109
Bell, James R., 324–326
Binary digit, 22
Binary number representation, 30–45
  excess-2^{m-1}, 42–43
  fixed-point, 42–43
  floating-point, 42–45
  fractions, 31
  normalized floating-point representation, 44–45
  ones' complement, 37–41, 43
  signed magnitude, 41, 43
  two's complement, 31–37, 42, 43
Binary search, 313–316
  analysis of, 315–316
Binary tree, 269–271
  arithmetic expressions represented as, 287–291
  relation to binary search, 314–315
  threaded, 295
  traversal of (*see* Traversal of binary trees)
Bit, 22
Bounds, 153–155
Branch (of tree), 268
Branch instructions, 27–28, 35–36, 65–69
Branch node, 269
Bubble sort, 300–302, 305
  analysis of, 302–304, 306–307
Buffer register, 207–209
Burks, Arthur, W., 52, 250
Burroughs B-5500:
  floating-point representation, 44
  stack instruction set, 295
Busy bit, 207, 209
Byte, 18, 58
Byte addressing, 18, 166–167

C (carry) condition code register, 34–36
Call by name, 194–195, 197–200
  ALGOL (replacement rule), 195, 197–200
Call by reference, 197, 200–201
Call by value, 194, 196, 200
  ALGOL, 196
  FORTRAN, 196
Carry, 31–36, 38–40
CDC 6600, 108
Chain-command bit, 230
Channel commands, 230–231
Church, Alonzo, 13
Circular list, 258, 275
Classification of computers, 104–109
  one-address, 106
  plus-one-address, 106–108
  three-address, 104–105
  two-address, 105
  zero-address, 108–109
Code (*see* Operation code)
Column major form of array storage, 156
Common data, 187–188
Complement, 31, 37, 95, 97
Compound statement in ALGOL, 346–347
Condition code, 27–28, 33–36, 65–69, 100
Conditional branch, 27–28, 35–36, 64–69
Conway, Melvin E., 201
Coroutines, 173, 179–186
  in ALGOL-like language, 350
  compared with subroutines, 179–180, 182–184
  implementation of, 181–182
  for input/output operations, 215–220, 234–249
  for multiple-pass algorithms, 184–186
Critical node, 330
Cyclical shift instructions, 97–103

D field, 69–70, 86–89
Data channel, 227–234, 256–257
  command-and-status register, 229
  word-count register, 229–230
Data packet, 165
Data structures (*see* Array; Link list; Push-down stack; Queue; Tree)
Davis, Martin, 13
Deferred address (*see* Indirect address)
Definite algorithm, 8
Definite instruction, 8
Degree of node, 269
Degree-*n* operators, 291
Deque (double-ended queue), 162
DESTINATION, 49–51, 84–86

Digital Equipment Corporation PDP-11 (*see* PDP-11)
Dijkstra, Edsger W., 250
Direct assignment, 115, 119–120
Direct-memory-access channel (*see* Data channel)
Displacement field, 69–70, 86–89
Division in System/360–370 computers, 61–63
Done bit, 207–209, 213
Double-ended queue (deque), 162
Double-precision operand, 101
Double-word,)8–60
  boundary, 58
  instructions, 64, 102
  register, 58, 60
Drum memory, 107
Dumey, Arnold I., 338

Eckert, J. Presper, 52
Effective address, 74, 80–82, 86–87 (*See also* Address modification)
Effective instruction, 6–7
Enable bit, 207–209
End-around carry, 38
  implementation of, 39–40
Endorder traversal, 272–273, 280
ENIAC, 52
Excess-2^{m-1} representation of numbers, 42–43
EXCLUSIVE OR, 94–97, 344
EXECUTE, 49–52, 86
Executive program, 70
Exponent, 42–45

FETCH, 49–51, 84
Fibonacci numbers, 296, 336–337
Finite algorithm, 7–8
Finite instruction, 7–8
Fixed-point representation of numbers, 42–43
Floating-point instructions, 64–65
Floating-point register, 58, 60, 64
Floating-point representation of numbers, 43–45
Floyd, Robert W., 307–310, 315, 338
Floyd's tree sort (*see* Tree sort)
Formal parameter, 187, 348
FORTRAN, 153, 156
  call by value, 196
Forward reference, 129
Foster, Caxton C., 327, 338
Four-color conjecture, 7–8
Fractions, representations of, 31
Free-storage list, 263–264
Full word, 58–59
  boundary, 58
  instructions, 60–63
  register, 58

Garbage collection, 263–264
General-purpose register, 58, 60, 63
  for addressing (*see* Base register; Index register)
GET, 158–162
Global data, 187–188
Goldstine, Herman H., 52, 250

Haley, A. C. D., 296
Halfword, 58–59
  boundary, 58
  instructions, 64
HALT, 9
Hash addressing, 316–326
  analysis of, 323–324
  linear quotient method, 325–326
  with linked lists, 317–321
  load factor, 323
  with secondary address generation, 321–326
  sequential search method, 321–324
Hash code, 317
Hash functions:
  modulo *N*, 320
  EXCLUSIVE OR, 320–321
HEAD, 158–162
Head, Turing machine, 8–9
Heap sort, 309–310 (*See also* Tree sort)
Height-balanced tree, 326–338
  analysis of, 331, 335–338
Hidden bit, 44
Hoare, C. A. R., 264, 338

I register, 49–51
IBM 704, 250
IBM System/360–370 (*see* System/360–370)
Immediate address mode, 84

Index (in array), 148
Index field, 69–70, 86–89
Index register, 80, 87–89
Indexing, 87–89
  in arrays, 148–158
Indirect address, 76–79, 83, 85–86
Indirection, addressing arrays by, 156–158
Infix notation, 288–289
Ingerman, Peter Z., 201
In-line parameter area, 188–191
Input/output, 205–212
  asynchronous control of, 234–249
  with multiple devices, 220–227
  in PDP-11, 208–227
  in System/360–370, 227–234
  timing problems, 211–212, 219–227
  use of interrupts, 212–220
  using data channels, 227–234
Input/output instructions, 209–212, 230–233
Institute for Advanced Study (IAS), 52
Instruction, 4–8
  definite, 8
  effective, 6–7
  encoding in PDP-11, 47–49
  encoding in System/360–370, 69–70
  execution of, 49–52
  finite, 7–8
  representation in memory, 46–52, 65–70
Instructions:
  PDP-11, table, 358–360
  System/360–370, table, 355–357
Interface, 206–207
Interior node, 269
Interrupt, 213
Interrupt enable bit, 213, 225
Interrupt processing, 212–219
  for asynchronous control of input/output, 234–249
  for multiple-device control, 220–227
  timing problems of, 211–212, 219–227
  (*See also* Data channel)
Interrupt system, 232
  PDP-11, 213–220, 247–249
  System/360–370, 231–233
Interrupt vector, 214–217, 226
IPL-V, 263–264

Kaman, Charles H., 325–326
Kent, William, 143
Knuth, Donald E., 264, 271, 295, 304, 336–338

Lampson, Butler, 250
Landis, E. M., 327
Leaf, 269
Left heavy node, 328
Left successor, 270–271
Leiner, A. L., 250
Level of node, 269
Library (*see* Program library)
Lin, Shen, 14
Linear quotient hash-addressing, 324–326
Link, 255
Link editor, 139
Link list, 254–258
  circular, 258
  one-way, 258
  threaded, 260–263
  two-way, 258–260, 275
LISP, 263, 278, 295
List (*see* Array; Link list)
Literal, 127–128
Load factor, 323
Load module, 139
Loader, 114, 133–139
  relocating, 133–139
  (*See also* Link editor)
LOCK, 244–247
Logical instructions, 94–97
Logical shift instructions, 97–103
  for data packing, 170
Łukasiewicz, Jan, 288–289, 295

McCarthy, John, 295
Machine status, 214, 226
Macro-instructions, 139–143
  nested, 142
Macros (*see* Macro-instructions)
Magnetic core, 21
Mantissa, 42–45
MAR (memory address register), 18–25
Mask:
  in branch instructions, 66–68
  for data packing, 167–170
Mauchly, John W., 52

Maurer, Ward Douglas, 324
MDR (memory data register), 18–25
Memory, 18–22
  bit-oriented, 22
  byte-oriented, 22
  drum, 107
  magnetic core, 21
  random-access, 18, 25
  semiconductor, 21–22
  word-oriented, 22
Memory address, 18–21
  (*See also* Address)
Memory allocation, 263–264
Memory organization:
  PDP-11, 18–22
  System/360–370, 58–59
Memory page, 92
Minsky, Marvin, 13
Minus zero, 41
Mock, Owen R., 250
Multiplication in System/360–370 computers, 61–63
Multiprogramming, 70, 94

N (negative) condition code register, 27–28, 33–36
Name of a variable, 194
  (*See also* Call by name)
Negative-number representation (*see* Binary number representation)
Nested macro, 142
Newell, Allen, 52, 109, 201, 263–264
Node, 268
Normalized floating-point representation of numbers, 44–45
NOT, 94–97, 344
Number, notion of, 30

Object module, 139
One-address computers, 106
Ones' complement representation of numbers, 37–41, 43
Operation code, 24
  encoding in PDP-11, 47–49
  encoding in System/360–370, 69–70
  reserved, 46
Operating system, 70
OR, 94–97, 344

Ordered tree, 269
  arithmetic expressions represented by, 270, 287–288
  binary tree representation, 277–279
  reconstruction from linear representation, 289–290
Overflow, 32–35, 38–40

P register, 23–27
Packed data, 273
Paging, 90–94
Paper-tape punch, 210–211
Paper-tape reader, 209–210
Parameter, 186–201
  (*See also* Call by name; Call by reference; Call by value)
Parameter area:
  in-line, 188–191
  push-down stack as, 192–194, 281–287
Parameters:
  actual, 187, 348
  formal, 187, 348
  passing techniques, 187–193
    in recursive subroutines, 281–287
  (*See also* Call by name; Call by reference; Call by value)
Parenthesis-free notation (*see* Postfix notation)
PDP-11:
  AC (accumulator), 23–25
  accumulator, 23–25, 79–80
  addition in, 33–35
  address modification, 74–86, 92–93
  address word, 75
  arithmetic shift instructions, 98–101
  assembler, 114–123, 129–132, 139–140
  bit-numbering convention, 46
  branch instructions, 27–29, 34–36
  buffer register, 207–209
  busy bit, 207, 209
  byte addressing, 166–167
  C (carry) condition code register, 34–36
  condition code, 27–28, 33–36, 39, 100
  coroutine calls in, 181
  cyclical shift instructions, 99–102

done bit, 207–209
enable bit, 207–209, 225
floating-point representation, 44–45
hidden bit, 44
index registers, 80, 83
input/output, 208–227
input/output instructions, 209–212
instructions, table, 358–360
interrupt enable bit, 213, 225
interrupt system, 213–220, 247–249
interrupt vector, 214–217, 226
logical instructions, 95–96
macro-instructions, 139–141, 142–143
memory organization, 18–22
N (negative) condition code register, 27–28, 33–36
ones' complement, 37–41, 43
overflow detection in, 33–35
P register (program counter), 23–27
processing unit organization, 22–25
processor status register, 213–217, 224, 247
program relocation in, 133–139
pseudo-instructions, 115–117, 119–122, 129–132, 136–137
ready bit, 207–209, 213
relative address, 74–76, 83
segmentation in, 93
software interrupt, 247–249
stack pointer, 174–175, 181, 213–215, 221–224, 247–248
status register, 207–209
subroutine calls in, 174–176
subtraction in, 36–37
V (overflow) condition code register, 33–36, 39
Z (zero) condition code register, 27–28, 33–36
Perlis, Alan J., 275, 295
Peterson, W. Wesley, 338
Plexes, 264
PL/I, 165, 194, 264
Plus-one-address computers, 106–108
Plus zero, 41
Pointer, 255
Polish notation, 288–289
POP, 163–165
Pop-up, 163
Postfix notation, 288–290, 295
Postorder traversal, 271–276, 280–281, 287–288
Predecessor, 268
Preorder traversal, 272–273, 280
Priority of interrupts, 224–226
Priority level, 224–227
Procedure, 186, 347–350
Processing unit, 17, 22–25
  PDP-11, 22–25
  System/360–370, 58–60, 70–71
Processor status register, 213–217, 224, 247
Program counter, 23
Program library, 138–139
Program location counter (PLC), 115, 117–120, 123, 124, 127
Program relocation (*see* Relocation)
Program status word, 70–71
Pseudo-instructions, 115–117, 119–128, 136–138
  processing of, 129–132, 136–138
PUSH, 163–165
Push-down, 163
Push-down stack, 162–165
  instruction set based on, 287–285
  for parameter passing, 192–194
    in recursive routines, 281–287
  used in tree traversal, 274–275
PUT, 158–162

Queue, 158–162
  double-ended, 162
  for input/output operations, 237–247
  linked-list representation, 256–257
Quicksort, 338

Radix, 30–33, 43–45, 53–54
Rado, Tibor, 14
Random access, 18, 25
Ranking, 302–304
  bubble sort analysis using, 303–304, 306–307
Ready bit, 207–209, 213
Recursion, 279
Recursive program, 279–287
Reentrant program, 179, 283
Register, 18
  for parameter passing, 188
Register addressing, 76, 85–86
Relative to the program counter, 75
Relative address, 74–76, 83

Relocating loader, 133–139
Relocation of programs, 76, 87–89, 91–93, 133–139
Replacement rule, 195, 197–200
REPLY, 206–208
REQUEST, 206–208
Reserved operation code, 46
Return address, 176
Right heavy node, 328
Right successor, 270–271
Rogers, Hartley, Jr., 13
Root, 268
Rooted tree, 267–269
  (*See also* Tree)
Rosen, Saul, 52
Ross, Douglas, T., 264
Rotate (*see* Cyclical shift instructions)
Row-major form of array storage, 156
RR instructions, 69–70
RX instructions, 69–70, 86–89

SAVE, 217–224
Searching techniques (*see* Binary search; Hash addressing; Height-balanced tree)
Secondary-operation code, 47
Segmentation, 93
Self-modifying program, 52, 179, 283
Semiconductor memory, 21–22
SHARE, 250
Shaw, Clifford J., 201, 263–264
Shift instructions, 97–104
SIGNAL, 239–244
Signed-magnitude representation of numbers, 41, 43
Simon, Herbert A., 201, 263–264
Single-operand instructions, 46–47, 50
Singleton, Richard C., 338
Software interrupt, 247–249
Sorting techniques (*see* Bubble sort; Quicksort; Tree sort)
SOURCE, 49–51, 84–85
Stack (*see* Push-down stack)
Stack instruction set, 287–295
Stack pointer, 174–175, 181, 213–215, 221–224, 247–248
State:
  of computer, 217–226
  Turing machine, 9
Status register, 207–209
Stirling approximation, 312
Stored program, 25–29, 52
Sublist, 265
Subroutine call, 176
Subroutine linkage register, 175
Subroutines, 173–179
  coroutines compared with, 179–180, 182–184
  for input/output operations, 215
  recursive, 280–287
  (*See also* Coroutines)
Subtree, 269
Successor, 268–269
Swift, Charles, J., 250
Symbol table, 117–122, 129–133, 136–137
System/360–370:
  address modification, 86–90, 93
  arithmetic instructions, 60–65
  arithmetic shift instructions, 102–104
  assembler, 124–128, 132, 141–142
  base registers, 87–90
  bit-numbering convention, 46
  branch instructions, 65–69
  channel commands, 230–231
  condition code, 65–69
  data types, 126
  double-word instructions, 64–65, 102
  floating-point instructions, 64–65
  floating-point registers, 58, 60, 64–65
  floating-point representation, 44–45
  general-purpose registers in, 58, 60, 63
  halfword instructions, 64
  index registers, 87–89
  input/output, 227–234
  input/output instructions, 230–233
  instructions, table, 355–357
  interrupt system, 231–233
  loader, 139
  logical instructions, 96–97
  logical shift instructions, 102–103
  macro-instructions, 139, 141–143
  memory organization, 58–59
  processing unit organization, 58–60, 70–71
  program relocation in, 134–139
  program status word, 70–71
  pseudo-instructions, 124–128, 132, 136, 138

representation of negative numbers, 61
RR instructions, 69–70
RX instructions, 69–70, 86–89
subroutine calls in, 177

TAIL, 158–162
Tape, Turing machine, 8–10, 19–20
Teletype, 211–212
Text-editing, 257
Thornton, Charles, 275, 295
Threaded list, 260–263
Threaded tree, 275–277, 295
Three-address computers, 104–105
Thunks, 201
Timing problems, input/output, 211–212, 219–227
Traversal of binary trees, 271–279, 295
endorder, 272–273, 280
postorder, 271–276, 280–281, 287–288
preorder, 272–273, 280
Tree, 267–271
AVE (*see* Height-balanced tree)
binary (*see* Binary tree)
height-balanced (*see* Height-balanced tree)
ordered (*see* Ordered tree)
relationship to recursive programs, 285–287
rooted, 268–269
threaded, 275–277, 295
Tree sort, 307–310, 315, 338
analysis of, 310–313
Turing, Alan M., 8, 13
Turing machine, 8–13
computer compared with, 17–20, 22–23, 25
head, 8–9
simulation of, 78–79
state, 9
tape, 8–9, 18–20
Turing machine program, 9–12
Two-address computers, 105
Two's complement representation of numbers, 31–37, 42, 43
Two-way linked list, 258–260, 275

Unary number representation, 10
Unconditional branch, 27–28, 67–68
UNLOCK, 244–247
UNSAVE, 217–224

V (overflow) condition code register, 33–36, 39
(*See also* Overflow)
Value of a variable, 194
(*See also* Call by value)
van Wijngaarden, A., 264
von Neumann, John, 52, 250

WAIT, 239–244
Wait loop, 208
Weight of digit, 32–33, 53–54
Williams, J. W. J., 309–310
Wirth, Niklaus, 250, 264
Word of memory, 20
Word address, 20

X field, 69–70, 86–89

Z (zero) condition code register, 27–28, 33–36
Zero-address computers, 108–109

0-07-061720-1